

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

Version 2.0.1

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

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

Generated on 2023/03/29 at 11:08:04 EDT

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

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

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

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

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

# Contents

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

48	3.4	Algebraic objects, operators and associated functions . . . . .	33
49	3.4.1	Operators . . . . .	34
50	3.4.2	Monoids . . . . .	39
51	3.4.3	Semirings . . . . .	39
52	3.5	Collections . . . . .	43
53	3.5.1	Scalars . . . . .	43
54	3.5.2	Vectors . . . . .	43
55	3.5.3	Matrices . . . . .	44
56	3.5.3.1	External matrix formats . . . . .	44
57	3.5.4	Masks . . . . .	44
58	3.6	Descriptors . . . . .	45
59	3.7	Fields . . . . .	46
60	3.7.1	String Handling . . . . .	49
61	3.7.2	Hints . . . . .	49
62	3.7.3	Input Types . . . . .	49
63	3.8	GrB_Info return values . . . . .	51
64	<b>4</b>	<b>Methods</b>	<b>53</b>
65	4.1	Context methods . . . . .	53
66	4.1.1	init: Initialize a GraphBLAS context . . . . .	53
67	4.1.2	finalize: Finalize a GraphBLAS context . . . . .	54
68	4.1.3	getVersion: Get the version number of the standard. . . . .	55
69	4.2	Object methods . . . . .	55
70	4.2.1	Get and Set methods . . . . .	56
71	4.2.1.1	get: Query the value of an object . . . . .	56
72	4.2.1.2	set: Set field of an object . . . . .	57
73	4.2.2	Algebra methods . . . . .	58
74	4.2.2.1	Type_new: Construct a new GraphBLAS (user-defined) type . . . .	58
75	4.2.2.2	UnaryOp_new: Construct a new GraphBLAS unary operator . . . .	59
76	4.2.2.3	BinaryOp_new: Construct a new GraphBLAS binary operator . . .	60
77	4.2.2.4	Monoid_new: Construct a new GraphBLAS monoid . . . . .	62

78	4.2.2.5	Semiring_new: Construct a new GraphBLAS semiring . . . . .	63
79	4.2.2.6	IndexUnaryOp_new: Construct a new GraphBLAS index unary op-	
80		erator [Scott: NEW CONTENT] . . . . .	64
81	4.2.3	Scalar methods . . . . .	66
82	4.2.3.1	Scalar_new: Construct a new scalar . . . . .	66
83	4.2.3.2	Scalar_dup: Construct a copy of a GraphBLAS scalar . . . . .	67
84	4.2.3.3	Scalar_clear: Clear/remove a stored value from a scalar . . . . .	68
85	4.2.3.4	Scalar_nvals: Number of stored elements in a scalar . . . . .	69
86	4.2.3.5	Scalar_setElement: Set the single element in a scalar . . . . .	70
87	4.2.3.6	Scalar_extractElement: Extract a single element from a scalar. . . .	71
88	4.2.4	Vector methods . . . . .	73
89	4.2.4.1	Vector_new: Construct new vector . . . . .	73
90	4.2.4.2	Vector_dup: Construct a copy of a GraphBLAS vector . . . . .	74
91	4.2.4.3	Vector_resize: Resize a vector . . . . .	75
92	4.2.4.4	Vector_clear: Clear a vector . . . . .	76
93	4.2.4.5	Vector_size: Size of a vector . . . . .	77
94	4.2.4.6	Vector_nvals: Number of stored elements in a vector . . . . .	77
95	4.2.4.7	Vector_build: Store elements from tuples into a vector . . . . .	78
96	4.2.4.8	Vector_setElement: Set a single element in a vector . . . . .	80
97	4.2.4.9	Vector_removeElement: Remove an element from a vector . . . . .	82
98	4.2.4.10	Vector_extractElement: Extract a single element from a vector. . . .	83
99	4.2.4.11	Vector_extractTuples: Extract tuples from a vector . . . . .	85
100	4.2.5	Matrix methods . . . . .	86
101	4.2.5.1	Matrix_new: Construct new matrix . . . . .	86
102	4.2.5.2	Matrix_dup: Construct a copy of a GraphBLAS matrix . . . . .	88
103	4.2.5.3	Matrix_diag: Construct a diagonal GraphBLAS matrix . . . . .	89
104	4.2.5.4	Matrix_resize: Resize a matrix . . . . .	90
105	4.2.5.5	Matrix_clear: Clear a matrix . . . . .	91
106	4.2.5.6	Matrix_nrows: Number of rows in a matrix . . . . .	92
107	4.2.5.7	Matrix_ncols: Number of columns in a matrix . . . . .	92
108	4.2.5.8	Matrix_nvals: Number of stored elements in a matrix . . . . .	93

109	4.2.5.9	Matrix_build: Store elements from tuples into a matrix . . . . .	94
110	4.2.5.10	Matrix_setElement: Set a single element in matrix . . . . .	96
111	4.2.5.11	Matrix_removeElement: Remove an element from a matrix . . . . .	98
112	4.2.5.12	Matrix_extractElement: Extract a single element from a matrix . . . .	99
113	4.2.5.13	Matrix_extractTuples: Extract tuples from a matrix . . . . .	101
114	4.2.5.14	Matrix_exportHint: Provide a hint as to which storage format might	
115		be most efficient for exporting a matrix . . . . .	103
116	4.2.5.15	Matrix_exportSize: Return the array sizes necessary to export a	
117		GraphBLAS matrix object . . . . .	104
118	4.2.5.16	Matrix_export: Export a GraphBLAS matrix to a pre-defined format	105
119	4.2.5.17	Matrix_import: Import a matrix into a GraphBLAS object . . . . .	107
120	4.2.5.18	Matrix_serializeSize: Compute the serialize buffer size . . . . .	109
121	4.2.5.19	Matrix_serialize: Serialize a GraphBLAS matrix. . . . .	110
122	4.2.5.20	Matrix_deserialize: Deserialize a GraphBLAS matrix. . . . .	111
123	4.2.6	Descriptor methods . . . . .	112
124	4.2.6.1	Descriptor_new: Create new descriptor . . . . .	112
125	4.2.6.2	Descriptor_set: Set content of descriptor . . . . .	113
126	4.2.7	free: Destroy an object and release its resources . . . . .	114
127	4.2.8	wait: Return once an object is either <i>complete</i> or <i>materialized</i> . . . . .	116
128	4.2.9	error: Retrieve an error string . . . . .	117
129	4.3	GraphBLAS operations . . . . .	118
130	4.3.1	mxm: Matrix-matrix multiply . . . . .	122
131	4.3.2	vxm: Vector-matrix multiply . . . . .	127
132	4.3.3	mxv: Matrix-vector multiply . . . . .	131
133	4.3.4	eWiseMult: Element-wise multiplication . . . . .	135
134	4.3.4.1	eWiseMult: Vector variant . . . . .	136
135	4.3.4.2	eWiseMult: Matrix variant . . . . .	140
136	4.3.5	eWiseAdd: Element-wise addition . . . . .	145
137	4.3.5.1	eWiseAdd: Vector variant . . . . .	146
138	4.3.5.2	eWiseAdd: Matrix variant . . . . .	150
139	4.3.6	extract: Selecting sub-graphs . . . . .	156

140	4.3.6.1	extract: Standard vector variant . . . . .	156
141	4.3.6.2	extract: Standard matrix variant . . . . .	160
142	4.3.6.3	extract: Column (and row) variant . . . . .	165
143	4.3.7	assign: Modifying sub-graphs . . . . .	170
144	4.3.7.1	assign: Standard vector variant . . . . .	170
145	4.3.7.2	assign: Standard matrix variant . . . . .	175
146	4.3.7.3	assign: Column variant . . . . .	181
147	4.3.7.4	assign: Row variant . . . . .	186
148	4.3.7.5	assign: Constant vector variant[Scott: NEW CONTENT] . . . . .	192
149	4.3.7.6	assign: Constant matrix variant[Scott: NEW CONTENT] . . . . .	197
150	4.3.8	apply: Apply a function to the elements of an object . . . . .	203
151	4.3.8.1	apply: Vector variant . . . . .	203
152	4.3.8.2	apply: Matrix variant . . . . .	208
153	4.3.8.3	apply: Vector-BinaryOp variants[Scott: NEW CONTENT] . . . . .	212
154	4.3.8.4	apply: Matrix-BinaryOp variants[Scott: NEW CONTENT] . . . . .	218
155	4.3.8.5	apply: Vector index unary operator variant[Scott: NEW CONTENT] . . . . .	224
156	4.3.8.6	apply: Matrix index unary operator variant[Scott: NEW CONTENT] . . . . .	229
157	4.3.9	select: . . . . .	234
158	4.3.9.1	select: Vector variant[Scott: NEW CONTENT] . . . . .	234
159	4.3.9.2	select: Matrix variant[Scott: NEW CONTENT] . . . . .	239
160	4.3.10	reduce: Perform a reduction across the elements of an object . . . . .	245
161	4.3.10.1	reduce: Standard matrix to vector variant . . . . .	245
162	4.3.10.2	reduce: Vector-scalar variant[Scott: NEW CONTENT] . . . . .	249
163	4.3.10.3	reduce: Matrix-scalar variant[Scott: NEW CONTENT] . . . . .	253
164	4.3.11	transpose: Transpose rows and columns of a matrix . . . . .	256
165	4.3.12	kroncker: Kronecker product of two matrices . . . . .	260
166	<b>5</b>	<b>Nonpolymorphic interface[Scott: NEW CONTENT]</b>	<b>267</b>
167	<b>A</b>	<b>Revision history</b>	<b>281</b>
168	<b>B</b>	<b>Non-opaque data format definitions</b>	<b>287</b>

169	B.1	GrB_Format: Specify the format for input/output of a GraphBLAS matrix. . . . .	287
170	B.1.1	GrB_CSR_FORMAT . . . . .	287
171	B.1.2	GrB_CSC_FORMAT . . . . .	288
172	B.1.3	GrB_COO_FORMAT . . . . .	288
173	<b>C</b>	<b>Examples</b>	<b>289</b>
174	C.1	Example: Level breadth-first search (BFS) in GraphBLAS . . . . .	290
175	C.2	Example: Level BFS in GraphBLAS using apply . . . . .	291
176	C.3	Example: Parent BFS in GraphBLAS . . . . .	292
177	C.4	Example: Betweenness centrality (BC) in GraphBLAS . . . . .	293
178	C.5	Example: Batched BC in GraphBLAS . . . . .	295
179	C.6	Example: Maximal independent set (MIS) in GraphBLAS . . . . .	297
180	C.7	Example: Counting triangles in GraphBLAS . . . . .	299



# List of Tables

2.1	Types of GraphBLAS opaque objects. . . . .	23
2.2	Methods that forced completion prior to GraphBLAS v2.0. . . . .	28
3.1	Enumeration literals and corresponding values input to various GraphBLAS methods. . . . .	32
3.2	Predefined GrB_Type values. . . . .	33
3.3	Operator input for relevant GraphBLAS operations. . . . .	34
3.4	Properties and recipes for building GraphBLAS algebraic objects. . . . .	35
3.5	Predefined unary and binary operators for GraphBLAS in C. . . . .	37
3.6	Predefined index unary operators for GraphBLAS in C. . . . .	38
3.7	Predefined monoids for GraphBLAS in C. . . . .	40
3.8	Predefined “true” semirings for GraphBLAS in C. . . . .	41
3.9	Other useful predefined semirings for GraphBLAS in C. . . . .	42
3.10	GrB_Format enumeration literals and corresponding values for matrix import and export methods. . . . .	44
3.11	Descriptor types and literals for fields and values. . . . .	47
3.12	Predefined GraphBLAS descriptors. . . . .	48
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, while All refers to all GraphBLAS objects. Global fields are denoted by Global. All fields may be read, some may be written (denoted by W), and some are hints (denoted by H) which may be ignored by the implementation. For * see 3.7 . . . . .	50
3.14	Descriptions of select <i>field, value</i> pairs listed in 3.13 . . . . .	51
3.15	Enumeration literals and corresponding values returned by GraphBLAS methods and operations. . . . .	52

206	4.1	A mathematical notation for the fundamental GraphBLAS operations supported in	
207		this specification. . . . .	119
208	5.1	Long-name, nonpolymorphic form of GraphBLAS methods. . . . .	267
209	5.2	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	268
210	5.3	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	269
211	5.4	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	270
212	5.5	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	271
213	5.6	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	272
214	5.7	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	273
215	5.8	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	274
216	5.9	Long-name, nonpolymorphic form of GraphBLAS methods (continued).[Scott: NEW	
217		CONTENT] . . . . .	275
218	5.10	Long-name, nonpolymorphic form of GraphBLAS methods (continued).[Scott: NEW	
219		CONTENT] . . . . .	276
220	5.11	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	277
221	5.12	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	278
222	5.13	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	279

# 223 List of Figures

224	3.1 Hierarchy of algebraic object classes in GraphBLAS. . . . .	43
225	4.1 Flowchart for the GraphBLAS operations. . . . .	120
226	B.1 Data layout for CSR format. . . . .	287
227	B.2 Data layout for CSC format. . . . .	288
228	B.3 Data layout for COO format. . . . .	288

## Acknowledgments

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

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

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

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

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

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

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

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

# Chapter 1

## Introduction

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

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

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

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

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

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

is not possible to express the signature of that method.

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

The remainder of this document is organized as follows:

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

## Chapter 2

# Basic concepts

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

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

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

## 2.1 Glossary

### 2.1.1 GraphBLAS API basic definitions

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

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

### 2.1.2 GraphBLAS objects and their structure

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



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

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

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

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

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

### 2.1.3 Algebraic structures used in the GraphBLAS

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

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

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

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

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

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

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

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

### 2.1.5 GraphBLAS methods: behaviors and error conditions

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

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

- 479 • *dimension compatible*: GraphBLAS objects (matrices and vectors) that are passed as param-  
480 eters to a GraphBLAS method are dimension (or shape) compatible if they have the correct  
481 number of dimensions and sizes for each dimension to satisfy the rules of the mathematical def-  
482 inition of the operation associated with the method. If any *dimension compatibility* rule above  
483 is violated, execution of the GraphBLAS method ends and the GrB\_DIMENSION\_MISMATCH  
484 error is returned.
- 485 • *domain compatible*: Two domains for which values from one domain can be cast to values in  
486 the other domain as per the rules of the C language. In particular, domains from Table 3.2  
487 are all compatible with each other, and a domain from a user-defined type is only compatible  
488 with itself. If any *domain compatibility* rule above is violated, execution of the GraphBLAS  
489 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

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

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

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

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

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

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

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

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

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

## 2.6 Error model

All GraphBLAS methods return a value of type `GrB_Info` (an enum) to provide information available to the system at the time the method returns. The returned value will be one of the defined values shown in Table 3.15. 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 3.15(a) are non-negative and include `GrB_SUCCESS` (a value of 0) and `GrB_NO_VALUE`.

An API error (listed in Table 3.15(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 3.15(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

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

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

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

750 After a call to any GraphBLAS method that modifies an opaque object, the program can re-  
751 trieve additional error information (beyond the error code returned by the method) though a call  
752 to the function `GrB_error()`, passing the method's output object as described in Section 4.2.9.  
753 The function returns a pointer to a NULL-terminated string, and the contents of that string are  
754 implementation-dependent. In particular, a null string (not a NULL pointer) is always a valid error  
755 string. `GrB_error()` is a thread-safe function, in the sense that multiple threads can call it simul-  
756 taneously and each will get its own error string back, referring to the object passed as an input  
757 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:

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

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

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

### 794 3.3 Types (domains)

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

---

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

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

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

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

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

---



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

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

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

### 3.4 Algebraic objects, operators and associated functions

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

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

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

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

Operation	Operator input
mxm, mxv, vxm	semiring
eWiseAdd	binary operator 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

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

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

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

### 3.4.1 Operators

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

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

---

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

(a) Properties of algebraic objects.

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

(b) Recipes for algebraic objects.

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

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

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

---

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

843 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  
844 domains,  $D_{out}$ ,  $D_{in_1}$ ,  $D_{in_2}$ , the domain of GraphBLAS indices, and an operation  $f_i : D_{in_1} \times I_{U64}^2 \times$   
845  $D_{in_2} \rightarrow D_{out}$  (where  $I_{U64}$  corresponds to the domain of a `GrB_Index`). For a given GraphBLAS  
846 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  $\mathbf{f}(F_i) = f_i$ .

847 User-defined operators can be created with calls to `GrB_UnaryOp_new`, `GrB_BinaryOp_new`, and  
848 `GrB_IndexUnaryOp_new`, respectively. See Section 4.2.2 for information on these methods. The  
849 GraphBLAS C API predefines a number of these operators. These are listed in Tables 3.5 and 3.6.  
850 Note that most entries in these tables represent a “family” of predefined operators for a set of  
851 different types represented by the  $T$ ,  $I$ , or  $F$  in their names. For example, the multiplicative  
852 inverse (`GrB_MINV_F`) function is only defined for floating-point types ( $F = \text{FP32}$  or  $\text{FP64}$ ). The  
853 division (`GrB_DIV_T`) function is defined for all types, but only if  $y \neq 0$  for integral and floating  
854 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 Name	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 `UINT $x$`  or `INT $x$` , and can be 32 or 64 in `FP $x$` .

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

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

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

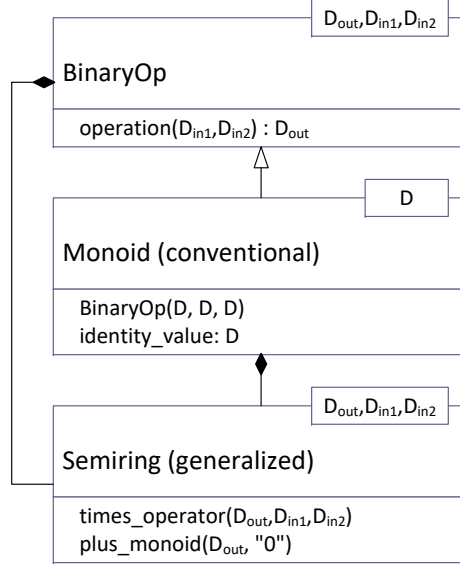


Figure 3.1: Hierarchy of algebraic object classes in GraphBLAS. GraphBLAS semirings consist of a conventional monoid with one domain for the addition function, and a binary operator with three domains for the multiplication function.

## 3.5 Collections

### 3.5.1 Scalars

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

### 3.5.2 Vectors

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

### 3.5.3 Matrices

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

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

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

#### 3.5.3.1 External matrix formats

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

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

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

### 3.5.4 Masks

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

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

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

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

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

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

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

## 3.6 Descriptors

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

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

For the purpose of constructing descriptors, the arguments of a method that can be modified

are identified by specific field names. The output parameter (typically the first parameter in a GraphBLAS method) is indicated by the field name, `GrB_OUTP`. The mask is indicated by the `GrB_MASK` field name. The input parameters corresponding to the input vectors and matrices are indicated by `GrB_INP0` and `GrB_INP1` in the order they appear in the signature of the GraphBLAS method. The descriptor is an opaque object and hence we do not define how objects of this type should be implemented. When referring to *(field, value)* pairs for a descriptor, however, we often use the informal notation `desc[GrB_Desc_Field].GrB_Desc_Value` without implying that a descriptor is to be implemented as an array of structures (in fact, field values can be used in conjunction with multiple values that are composable). We summarize all types, field names, and values used with descriptors in Table 3.11.

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

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

GraphBLAS specifies all of the valid combinations of (field, value) pairs as predefined descriptors. Their identifiers and the corresponding set of (field, value) pairs for that identifier are shown in Table 3.12.

## 3.7 Fields

All GraphBLAS objects and implementations contain fields like those in the descriptor, which provide information to users and allow setting runtime parameters and hints. All GraphBLAS objects are required to implement the `get` and `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 *value, field* pairs available for each object are defined in 3.13, although implementations may add `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_NAME` field is a special case regarding writability. All objects which have a `GrB_NAME` field default to an empty string, `GrB_NAMESIZE` will be 0. Collections and `GrB_Descriptors` may have their `GrB_NAME` set at any time. Algebraic objects and `GrB_Types` may only have their

---

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
GrB_Descriptor	Type of a GraphBLAS descriptor object.
GrB_Desc_Field	The descriptor field enumeration.
GrB_Desc_Value	The descriptor value enumeration.

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

Field Name	Value	Description
GrB_OUTP	0	Field name for the output GraphBLAS object.
GrB_MASK	1	Field name for the mask GraphBLAS object.
GrB_INP0	2	Field name for the first input GraphBLAS object.
GrB_INP1	3	Field name for the second input GraphBLAS object.

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

Value Name	Value	Description
(reserved)	0	Unused
GrB_REPLACE	1	Clear the output object before assigning computed values.
GrB_COMP	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.
GrB_TRAN	3	Use the transpose of the associated object.
GrB_STRUCTURE	4	The write mask is constructed from the structure (pattern of stored values) of the associated object. The stored values are not examined.

---

Table 3.12: 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



1005 GrB\_NAME set once to a globally unique value. Attempting to set this field after it has already  
1006 been set will return a GrB\_ALREADY\_SET error code.

1007 The GrB\_Field enumeration is defined by the values in Table 3.13, and selected values are described  
1008 in Table 3.14.

### 1009 3.7.1 String Handling

1010 When the input to GrB\_<OBJ>\_set is a char\* the input array is null terminated. The GraphBLAS  
1011 implementation must copy this array into internal data structures.

1012 When a char\* is the output argument of GrB\_<OBJ>\_get the user must preallocate a properly  
1013 sized buffer. The returned string is null terminated, so the buffer must be at least 1 larger than the  
1014 size of the string. For instance the buffer for GrB\_NAME must be of length GrB\_NAMESIZE + 1.

### 1015 3.7.2 Hints

1016 Several fields are *hints* (marked H in Table 3.13). A GraphBLAS implementation is free to ignore  
1017 a hint and return GrB\_SUCCESS. For instance GrB\_NTHREADS might be ignored by a sequential  
1018 GraphBLAS implementation.

### 1019 3.7.3 Input Types

1020 By default arguments are passed in a GrB\_Scalar, only char\* and GraphBLAS objects like GrB\_Type  
1021 are passed in directly. Enums are passed as GrB\_INT32 GrBScalars.

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, while All refers to all GraphBLAS objects. Global fields are denoted by Global. All fields may be read, some may be written (denoted by W), and some are hints (denoted by H) which may be ignored by the implementation. For \* see 3.7

Field Name	W	H	Value	Implementing Objects	Type
GrB_OUTP	W	—	0	GrB_Descriptor	GrB_Desc_Value
GrB_MASK	W	—	1	GrB_Descriptor	GrB_Desc_Value
GrB_INP0	W	—	2	GrB_Descriptor	GrB_Desc_Value
GrB_INP1	W	—	3	GrB_Descriptor	GrB_Desc_Value
GrB_NAMESIZE	—	—	10	All, Global	GrB_INT32
GrB_NAME	*	—	11	All	Null terminated char* of size GrB_NAMESIZE
GrB_LIBRARY_VER_MAJOR	—	—	101	Global	GrB_INT32
GrB_LIBRARY_VER_MINOR	—	—	101	Global	GrB_INT32
GrB_LIBRARY_VER_PATCH	—	—	101	Global	GrB_INT32
GrB_API_VER_MAJOR	—	—	102	Global	GrB_INT32
GrB_API_VER_MINOR	—	—	102	Global	GrB_INT32
GrB_API_VER_PATCH	—	—	102	Global	GrB_INT32
GrB_BLOCKING_MODE	—	—	103	Global	GrB_Mode
GrB_NTHREADS	W	H	104	Global, GrB_Descriptor	GrB_INT32
GrB_STORAGE_ORIENTATION_HINT	W	H	200	Global, Collection	GrB_ROWMAJOR, GrB_COLMAJOR
GrB_STORAGE_FORMAT_HINT	W	H	201	Collection	GrB_Format
GrB_ELTYPE	—	—	202	Collection	GrB_TypeEnum
GrB_INPUT1TYPE	—	—	300	Algebraic	GrB_TypeEnum
GrB_INPUT2TYPE	—	—	301	Algebraic	GrB_TypeEnum
GrB_OUTPUTTYPE	—	—	302	Algebraic	GrB_TypeEnum
GrB_BINARYOP	—	—	303	GrB_Monoid, GrB_Semiring	GrB_BinaryOpEnum
GrB_MONOID	—	—	304	GrB_Semiring	GrB_MonoidEnum
GrB_ENUM	—	—	304	Algebraic, Collection	GrB_INT32

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

Field Name	Description
GrB_NAMESIZE	The length of the GrB_NAME for a particular object.
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_NTHREADS	The number of threads for the library to use for a function call, may be ignored by the library.
GrB_STORAGE_ORIENTATION_HINT	Hint to the library that a collection is best stored in a row (lexicographic) or column (colexicographic) major format.
GrB_STORAGE_FORMAT_HINT	Hint to the library that it should use a specific storage format.
GrB_ELTYPE	The element type of a collection.
GrB_INPUT1TYPE	The type of the first argument to an operator.
GrB_INPUT2TYPE	The type of the second argument to an operator.
GrB_OUTPUTTYPE	The type of the output of an operator.
GrB_BINARYOP	The binary operator of a semiring or monoid.
GrB_MONOID	The monoid of a semiring.

### 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 3.15.

Table 3.15: 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`.

## 1044 **Return Values**

1045                   GrB\_SUCCESS operation completed successfully.

1046                   GrB\_PANIC unknown internal error.

1047                   GrB\_INVALID\_VALUE invalid mode specified, or method called multiple times.

## 1048 **Description**

1049 The init method creates and initializes a GraphBLAS C API context. The argument to GrB\_init  
1050 defines the mode for the context. The two available modes are:

- 1051       • GrB\_BLOCKING: In this mode, each method in a sequence returns after its computations have  
1052        completed and output arguments are available to subsequent statements in an application.  
1053        When executing in GrB\_BLOCKING mode, the methods execute in program order.
- 1054       • GrB\_NONBLOCKING: In this mode, methods in a sequence may return after arguments in  
1055        the method have been tested for dimension and domain compatibility within the method  
1056        but potentially before their computations complete. Output arguments are available to sub-  
1057        sequent GraphBLAS methods in an application. When executing in GrB\_NONBLOCKING  
1058        mode, the methods in a sequence may execute in any order that preserves the mathematical  
1059        result defined by the sequence.

1060 An application can only create one context per execution instance. An application may only call  
1061 GrB\_Init once. Calling GrB\_Init more than once results in undefined behavior.

### 1062 **4.1.2 finalize: Finalize a GraphBLAS context**

1063 Terminates and frees any internal resources created to support the GraphBLAS C API context.

## 1064 **C Syntax**

1065                   GrB\_Info GrB\_finalize();

## 1066 **Return Values**

1067                   GrB\_SUCCESS operation completed successfully.

1068                   GrB\_PANIC unknown internal error.

## 1069 **Description**

1070 The `finalize` method terminates and frees any internal resources created to support the GraphBLAS  
1071 C API context. `GrB_finalize` may only be called after a context has been initialized by calling  
1072 `GrB_init`, or else undefined behavior occurs. After `GrB_finalize` has been called to finalize a Graph-  
1073 BLAS context, calls to any GraphBLAS methods, including `GrB_finalize`, will result in undefined  
1074 behavior.

### 1075 **4.1.3 getVersion: Get the version number of the standard.**

1076 Query the library for the version number of the standard that this library implements.

## 1077 **C Syntax**

```
1078         GrB_Info GrB_getVersion(unsigned int *version,  
1079                                unsigned int *subversion);
```

## 1080 **Parameters**

1081 version (OUT) On successful return will hold the value of the major version number.

1082 version (OUT) On successful return will hold the value of the subversion number.

## 1083 **Return Values**

1084 GrB\_SUCCESS operation completed successfully.

1085 GrB\_PANIC unknown internal error.

## 1086 **Description**

1087 The `getVersion` method is used to query the major and minor version number of the GraphBLAS  
1088 C API specification that the library implements at runtime. To support compile time queries the  
1089 following two macros shall also be defined by the library.

```
1090         #define GRB_VERSION      2  
1091         #define GRB_SUBVERSION  0
```

## 1092 **4.2 Object methods**

1093 This section describes methods that setup and operate on GraphBLAS opaque objects but are not  
1094 part of the the GraphBLAS math specification.

## 1095 4.2.1 Get and Set methods

1096 The methods in this section query and, optionally, set internal fields of GraphBLAS objects.

### 1097 4.2.1.1 get: Query the value of an object

#### 1098 C Syntax

```
1099     GrB_Info GrB_<OBJ>_get(GrB_<OBJ> o, GrB_Field field, <type> value);
1100
1101     GrB_Info GrB_Scalar_get(GrB_Scalar s, GrB_Field field, <type> value);
1102     GrB_Info GrB_Vector_get(GrB_Vector v, GrB_Field field, <type> value);
1103     GrB_Info GrB_Matrix_get(GrB_Matrix A, GrB_Field field, <type> value);
1104
1105     GrB_Info GrB_UnaryOp_get(GrB_UnaryOp op, GrB_Field field, <type> value);
1106     GrB_Info GrB_IndexUnaryOp_get(GrB_IndexUnaryOp op, GrB_Field field, <type> value);
1107     GrB_Info GrB_BinaryOp_get(GrB_BinaryOp op, GrB_Field field, <type> value);
1108     GrB_Info GrB_Monoid_get(GrB_Monoid op, GrB_Field field, <type> value);
1109     GrB_Info GrB_Semiring_get(GrB_Semiring op, GrB_Field field, <type> value);
1110
1111     GrB_Info GrB_Descriptor_get(GrB_Descriptor desc, GrB_Field field, <type> value);
1112     GrB_Info GrB_Type_get(GrB_Type type, GrB_Field field, <type> value);
1113
1114     GrB_Info GrB_Global_get(GrB_Field field, <type> value);
```

#### 1115 Parameters

1116 OBJ (IN) An existing, valid GraphBLAS object which is being queried.

1117 field (IN) The field being queried.

1118 value (OUT) A pointer to or GrB\_Scalar containing a value whose type is dependent on  
1119 field which will be filled with the current value of the field. type may be a char\*,  
1120 GrB\_Scalar, GrB\_INT32 or void\*.

#### 1121 Return Value

1122 GrB\_SUCCESS The method completed successfully.

1123 GrB\_PANIC unknown internal error.

1124 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1125 GrB\_UNINITIALIZED\_OBJECT the desc parameter has not been initialized by a call to new.

1126 GrB\_INVALID\_VALUE invalid value set on the field or invalid field.



## 1127 Description

1128 Queries a field of an existing GraphBLAS object. The type of the argument is uniquely determined  
1129 by field. Fields marked as hints in Table 3.13 will return the hint when queried, not the true internal  
1130 value.

### 1131 4.2.1.2 set: Set field of an object

1132 Set the content for a field for an existing GraphBLAS object.

## 1133 C Syntax

```
1134     GrB_Info GrB_<OBJ>_set(GrB_<OBJ> o, GrB_Field field, <type> value);
1135
1136     GrB_Info GrB_Scalar_set(GrB_Scalar s, GrB_Field field, <type> value);
1137     GrB_Info GrB_Vector_set(GrB_Vector v, GrB_Field field, <type> value);
1138     GrB_Info GrB_Matrix_set(GrB_Matrix A, GrB_Field field, <type> value);
1139
1140     GrB_Info GrB_UnaryOp_set(GrB_UnaryOp op, GrB_Field field, <type> value);
1141     GrB_Info GrB_IndexUnaryOp_set(GrB_IndexUnaryOp op, GrB_Field field, <type> value);
1142     GrB_Info GrB_BinaryOp_set(GrB_BinaryOp op, GrB_Field field, <type> value);
1143     GrB_Info GrB_Monoid_set(GrB_Monoid op, GrB_Field field, <type> value);
1144     GrB_Info GrB_Semiring_set(GrB_Semiring op, GrB_Field field, <type> value);
1145
1146     GrB_Info GrB_Descriptor_set(GrB_Descriptor op, GrB_Field field, <type> value);
1147     GrB_Info GrB_Type_set(GrB_Type op, GrB_Field field, <type> value);
1148
1149     GrB_Info GrB_Global_set(GrB_Field field, <type> value);
```

## 1150 Parameters

1151 OBJ (IN) The GraphBLAS object which is having field set.

1152 field (IN) The field being set.

1153 value (IN) A value whose type is dependent on field or a GrB\_Scalar. type  
1154 may be a char\*, GrB\_Scalar, GrB\_INT32 or void\*.

## 1155 Return Values

1156 GrB\_SUCCESS The method completed successfully.

1157 GrB\_PANIC unknown internal error.

1158 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1159 `GrB_UNINITIALIZED_OBJECT` the `desc` parameter has not been initialized by a call to `new`.  
 1160 `GrB_INVALID_VALUE` invalid value set on the field, invalid field, or field is read-only.  
 1161 `GrB_ALREADY_SET` this field has already been set, and may only be set once.

## 1162 **Description**

1163 Set a field of `OBJ` to a new value.

## 1164 **4.2.2 Algebra methods**

### 1165 **4.2.2.1 `Type_new`: Construct a new GraphBLAS (user-defined) type**

1166 Creates a new user-defined GraphBLAS type. This type can then be used to create new operators,  
 1167 monoids, semirings, vectors and matrices.

## 1168 **C Syntax**

```
1169      GrB_Info GrB_Type_new(GrB_Type  *utype,
1170                          size_t      sizeof(ctype));
```

## 1171 **Parameters**

1172 `utype` (INOUT) On successful return, contains a handle to the newly created user-defined  
 1173 GraphBLAS type object.

1174 `ctype` (IN) A C type that defines the new GraphBLAS user-defined type.

## 1175 **Return Values**

1176 `GrB_SUCCESS` operation completed successfully.

1177 `GrB_PANIC` unknown internal error.

1178 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1179 `GrB_NULL_POINTER` `utype` pointer is `NULL`.

## 1180 **Description**

1181 Given a C type `ctype`, the `Type_new` method returns in `utype` a handle to a new GraphBLAS type  
 1182 that is equivalent to the C type. Variables of this `ctype` must be a struct, union, or fixed-size array.  
 1183 In particular, given two variables, `src` and `dst`, of type `ctype`, the following operation must be a  
 1184 valid way to copy the contents of `src` to `dst`:

1185                   memcpy(&dst, &src, sizeof(ctype))

1186 A new, user-defined type `utype` should be destroyed with a call to `GrB_free(utype)` when no longer  
1187 needed.

1188 It is not an error to call this method more than once on the same variable; however, the handle to  
1189 the previously created object will be overwritten.

#### 1190 4.2.2.2 UnaryOp\_new: Construct a new GraphBLAS unary operator

1191 Initializes a new GraphBLAS unary operator with a specified user-defined function and its types  
1192 (domains).

### 1193 C Syntax

```
1194     GrB_Info GrB_UnaryOp_new(GrB_UnaryOp *unary_op,  
1195                             void          (*unary_func)(void*, const void*),  
1196                             GrB_Type      d_out,  
1197                             GrB_Type      d_in);
```

### 1198 Parameters

1199     `unary_op` (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1200     unary operator object.

1201     `unary_func` (IN) a pointer to a user-defined function that takes one input parameter of `d_in`'s  
1202     type and returns a value of `d_out`'s type, both passed as `void` pointers. Specifically  
1203     the signature of the function is expected to be of the form:

```
1204                   void func(void *out, const void *in);  
1205
```

1206     `d_out` (IN) The `GrB_Type` of the return value of the unary operator being created. Should  
1207     be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-  
1208     BLAS type.

1209     `d_in` (IN) The `GrB_Type` of the input argument of the unary operator being created.  
1210     Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1211     GraphBLAS type.

### 1212 Return Values

1213     `GrB_SUCCESS` operation completed successfully.

1214     `GrB_PANIC` unknown internal error.

1215       GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1216 GrB\_UNINITIALIZED\_OBJECT any GrB\_Type parameter (for user-defined types) has not been ini-  
1217                               tialized by a call to GrB\_Type\_new.

1218       GrB\_NULL\_POINTER unary\_op or unary\_func pointers are NULL.

## 1219 Description

1220 The UnaryOp\_new method creates a new GraphBLAS unary operator

1221        $f_u = \langle \mathbf{D}(d\_out), \mathbf{D}(d\_in), unary\_func \rangle$

1222 and returns a handle to it in unary\_op.

1223 The implementation of unary\_func must be such that it works even if the d\_out and d\_in arguments  
1224 are aliased. In other words, for all invocations of the function:

1225       unary\_func(out, in);

1226 the value of out must be the same as if the following code was executed:

```
1227       D(d_in) *tmp = malloc(sizeof(D(d_in)));  
1228       memcpy(tmp, in, sizeof(D(d_in)));  
1229       unary_func(out, tmp);  
1230       free(tmp);
```

1231 It is not an error to call this method more than once on the same variable; however, the handle to  
1232 the previously created object will be overwritten.

### 1233 4.2.2.3 BinaryOp\_new: Construct a new GraphBLAS binary operator

1234 Initializes a new GraphBLAS binary operator with a specified user-defined function and its types  
1235 (domains).

## 1236 C Syntax

```
1237       GrB_Info GrB_BinaryOp_new(GrB_BinaryOp *binary_op,  
1238                               void           (*binary_func)(void*,  
1239                                               const void*,  
1240                                               const void*),  
1241                               GrB_Type       d_out,  
1242                               GrB_Type       d_in1,  
1243                               GrB_Type       d_in2);
```

## 1244 Parameters

1245     **binary\_op** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1246     binary operator object.

1247     **binary\_func** (IN) A pointer to a user-defined function that takes two input parameters of types  
1248     **d\_in1** and **d\_in2** and returns a value of type **d\_out**, all passed as **void** pointers.  
1249     Specifically the signature of the function is expected to be of the form:

1250             **void func(void \*out, const void \*in1, const void \*in2);**  
1251

1252     **d\_out** (IN) The **GrB\_Type** of the return value of the binary operator being created. Should  
1253     be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-  
1254     BLAS type.

1255     **d\_in1** (IN) The **GrB\_Type** of the left hand argument of the binary operator being created.  
1256     Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1257     GraphBLAS type.

1258     **d\_in2** (IN) The **GrB\_Type** of the right hand argument of the binary operator being cre-  
1259     ated. Should be one of the predefined GraphBLAS types in Table 3.2, or a user-  
1260     defined GraphBLAS type.

## 1261 Return Values

1262     **GrB\_SUCCESS** operation completed successfully.

1263     **GrB\_PANIC** unknown internal error.

1264     **GrB\_OUT\_OF\_MEMORY** not enough memory available for operation.

1265     **GrB\_UNINITIALIZED\_OBJECT** the **GrB\_Type** (for user-defined types) has not been initialized by a  
1266     call to **GrB\_Type\_new**.

1267     **GrB\_NULL\_POINTER** **binary\_op** or **binary\_func** pointer is **NULL**.

## 1268 Description

1269     The **BinaryOp\_new** methods creates a new GraphBLAS binary operator

1270      $f_b = \langle \mathbf{D}(\mathbf{d\_out}), \mathbf{D}(\mathbf{d\_in1}), \mathbf{D}(\mathbf{d\_in2}), \mathbf{binary\_func} \rangle$

1271     and returns a handle to it in **binary\_op**.

1272     The implementation of **binary\_func** must be such that it works even if any of the **d\_out**, **d\_in1**, and  
1273     **d\_in2** arguments are aliased to each other. In other words, for all invocations of the function:

1274        `binary_func(out,in1,in2);`

1275    the value of `out` must be the same as if the following code was executed:

```
1276        D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));
1277        D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));
1278        memcpy(tmp1,in1,sizeof(D(d_in1)));
1279        memcpy(tmp2,in2,sizeof(D(d_in2)));
1280        binary_func(out,tmp1,tmp2);
1281        free(tmp2);
1282        free(tmp1);
```

1283    It is not an error to call this method more than once on the same variable; however, the handle to  
1284    the previously created object will be overwritten.

#### 1285    **4.2.2.4    Monoid\_new: Construct a new GraphBLAS monoid**

1286    Creates a new monoid with specified binary operator and identity value.

#### 1287    **C Syntax**

```
1288        GrB_Info GrB_Monoid_new(GrB_Monoid    *monoid,  
1289                                GrB_BinaryOp    binary_op,  
1290                                <type>        identity);
```

#### 1291    **Parameters**

1292        `monoid` (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1293        monoid object.

1294        `binary_op` (IN) An existing GraphBLAS associative binary operator whose input and output  
1295        types are the same.

1296        `identity` (IN) The value of the identity element of the monoid. Must be the same type as  
1297        the type used by the `binary_op` operator.

#### 1298    **Return Values**

1299        `GrB_SUCCESS` operation completed successfully.

1300        `GrB_PANIC` unknown internal error.

1301        `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1302 GrB\_UNINITIALIZED\_OBJECT the GrB\_BinaryOp (for user-defined operators) has not been initial-  
1303 ized by a call to GrB\_BinaryOp\_new.

1304 GrB\_NULL\_POINTER monoid pointer is NULL.

1305 GrB\_DOMAIN\_MISMATCH all three argument types of the binary operator and the type of the  
1306 identity value are not the same.

## 1307 Description

1308 The Monoid\_new method creates a new monoid  $M = \langle \mathbf{D}(\text{binary\_op}), \text{binary\_op}, \text{identity} \rangle$  and re-  
1309 turns a handle to it in monoid.

1310 If binary\_op is not associative, the results of GraphBLAS operations that require associativity of  
1311 this monoid will be undefined.

1312 It is not an error to call this method more than once on the same variable; however, the handle to  
1313 the previously created object will be overwritten.

### 1314 4.2.2.5 Semiring\_new: Construct a new GraphBLAS semiring

1315 Creates a new semiring with specified domain, operators, and elements.

## 1316 C Syntax

```
1317 GrB_Info GrB_Semiring_new(GrB_Semiring *semiring,  
1318                           GrB_Monoid   add_op,  
1319                           GrB_BinaryOp  mul_op);
```

## 1320 Parameters

1321 semiring (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1322 semiring.

1323 add\_op (IN) An existing GraphBLAS commutative monoid that specifies the addition op-  
1324 erator and its identity.

1325 mul\_op (IN) An existing GraphBLAS binary operator that specifies the semiring's multi-  
1326 plication operator. In addition, mul\_op's output domain,  $\mathbf{D}_{out}(\text{mul\_op})$ , must be  
1327 the same as the add\_op's domain  $\mathbf{D}(\text{add\_op})$ .

## 1328 Return Values

1329 GrB\_SUCCESS operation completed successfully.

1330                   GrB\_PANIC unknown internal error.

1331           GrB\_OUT\_OF\_MEMORY not enough memory available for this method to complete.

1332 GrB\_UNINITIALIZED\_OBJECT the `add_op` (for user-define monoids) object has not been initialized  
1333                                   with a call to `GrB_Monoid_new` or the `mul_op` (for user-defined  
1334                                   operators) object has not been not been initialized by a call to  
1335                                   `GrB_BinaryOp_new`.

1336           GrB\_NULL\_POINTER semiring pointer is NULL.

1337           GrB\_DOMAIN\_MISMATCH the output domain of `mul_op` does not match the domain of the  
1338                                   `add_op` monoid.

## 1339 Description

1340 The `Semiring_new` method creates a new semiring:

$$1341 \quad S = \langle \mathbf{D}_{out}(\text{mul\_op}), \mathbf{D}_{in_1}(\text{mul\_op}), \mathbf{D}_{in_2}(\text{mul\_op}), \text{add\_op}, \text{mul\_op}, \mathbf{0}(\text{add\_op}) \rangle$$

1342 and returns a handle to it in `semiring`. Note that  $\mathbf{D}_{out}(\text{mul\_op})$  must be the same as  $\mathbf{D}(\text{add\_op})$ .  
1343 If `add_op` is not commutative, then GraphBLAS operations using this semiring will be undefined.  
1344 It is not an error to call this method more than once on the same variable; however, the handle to  
1345 the previously created object will be overwritten.

### 1346 4.2.2.6 IndexUnaryOp\_new: Construct a new GraphBLAS index unary operator [Scott: 1347                                   NEW CONTENT]

1348 Initializes a new GraphBLAS index unary operator with a specified user-defined function and its  
1349 types (domains).

## 1350 C Syntax

```

1351 GrB_Info GrB_IndexUnaryOp_new(GrB_IndexUnaryOp  *index_unary_op,
1352                               void (*index_unary_func)(void*,
1353                               const void*,
1354                               GrB_Index,
1355                               GrB_Index,
1356                               const void*),
1357                               GrB_Type          d_out,
1358                               GrB_Type          d_in1,
1359                               GrB_Type          d_in2);

```



## 1360 Parameters

1361 `index_unary_op` (INOUT) On successful return, contains a handle to the newly created Graph-  
1362 BLAS index unary operator object.

1363 `index_unary_func` (IN) A pointer to a user-defined function that takes input parameters of types  
1364 `d_in1`, `GrB_Index`, `GrB_Index` and `d_in2` and returns a value of type `d_out`. Ex-  
1365 cept for the `GrB_Index` parameters, all are passed as `void` pointers. Specifically  
1366 the signature of the function is expected to be of the form:

```
1367         void func(void      *out,  
1368                   const void *in1,  
1369                   GrB_Index  row_index,  
1370                   GrB_Index  col_index,  
1371                   const void *in2);
```

1373 `d_out` (IN) The `GrB_Type` of the return value of the index unary operator being created.  
1374 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1375 GraphBLAS type.

1376 `d_in1` (IN) The `GrB_Type` of the first input argument of the index unary operator being  
1377 created and corresponds to the stored values of the `GrB_Vector` or `GrB_Matrix`  
1378 being operated on. Should be one of the predefined GraphBLAS types in Ta-  
1379 ble 3.2, or a user-defined GraphBLAS type.

1380 `d_in2` (IN) The `GrB_Type` of the last input argument of the index unary operator be-  
1381 ing created and corresponds to a scalar provided by the GraphBLAS operation  
1382 that uses this operator. Should be one of the predefined GraphBLAS types in  
1383 Table 3.2, or a user-defined GraphBLAS type.

## 1384 Return Values

1385 `GrB_SUCCESS` operation completed successfully.

1386 `GrB_PANIC` unknown internal error.

1387 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1388 `GrB_UNINITIALIZED_OBJECT` the `GrB_Type` (for user-defined types) has not been initialized by a  
1389 call to `GrB_Type_new`.

1390 `GrB_NULL_POINTER` `index_unary_op` or `index_unary_func` pointer is `NULL`.

## 1391 Description

1392 The `IndexUnaryOp_new` methods creates a new GraphBLAS index unary operator

1393  $f_i = \langle \mathbf{D}(d\_out), \mathbf{D}(d\_in1), \mathbf{D}(GrB\_Index), \mathbf{D}(GrB\_Index), \mathbf{D}(d\_in2), index\_unary\_func) \rangle$

1394 and returns a handle to it in `index_unary_op`.

1395 The implementation of `index_unary_func` must be such that it works even if any of the `d_out`,  
 1396 `d_in1`, and `d_in2` arguments are aliased to each other. In other words, for all invocations of the  
 1397 function:

1398 `index_unary_func(out,in1,row_index,col_index,n,in2);`

1399 the value of `out` must be the same as if the following code was executed (shown here for matrices):

```
1400 GrB_Index row_index = ...;
1401 GrB_Index col_index = ...;
1402 D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));
1403 D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));
1404 memcpy(tmp1,in1,sizeof(D(d_in1)));
1405 memcpy(tmp2,in2,sizeof(D(d_in2)));
1406 index_unary_func(out,tmp1,row_index,col_index,tmp2);
1407 free(tmp2);
1408 free(tmp1);
```

1409 It is not an error to call this method more than once on the same variable; however, the handle to  
 1410 the previously created object will be overwritten.

## 1411 4.2.3 Scalar methods

### 1412 4.2.3.1 Scalar\_new: Construct a new scalar

1413 Creates a new empty scalar with specified domain.

## 1414 C Syntax

```
1415 GrB_Info GrB_Scalar_new(GrB_Scalar *s,
1416                          GrB_Type d);
```

## 1417 Parameters

1418 **s** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
 1419 scalar.

1420 **d** (IN) The type corresponding to the domain of the scalar being created. Can be  
 1421 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
 1422 GraphBLAS type.

## 1423 Return Values

1424           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1425                       blocking mode, this indicates that the API checks for the input  
1426                       arguments passed successfully. Either way, output scalar `s` is ready  
1427                       to be used in the next method of the sequence.

1428           GrB\_PANIC Unknown internal error.

1429           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1430                       GraphBLAS objects (input or output) is in an invalid state caused  
1431                       by a previous execution error. Call `GrB_error()` to access any error  
1432                       messages generated by the implementation.

1433           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1434 GrB\_UNINITIALIZED\_OBJECT The `GrB_Type` object has not been initialized by a call to `GrB_Type_new`  
1435                       (needed for user-defined types).

1436           GrB\_NULL\_POINTER The `s` pointer is NULL.

## 1437 Description

1438 Creates a new GraphBLAS scalar `s` of domain `D(d)` and empty `L(s)`. The method returns a handle  
1439 to the new scalar in `s`.

1440 It is not an error to call this method more than once on the same variable; however, the handle to  
1441 the previously created object will be overwritten.

### 1442 4.2.3.2 Scalar\_dup: Construct a copy of a GraphBLAS scalar

1443 Creates a new scalar with the same domain and contents as another scalar.

## 1444 C Syntax

```
1445           GrB_Info GrB_Scalar_dup(GrB_Scalar           *t,  
1446                                   const GrB_Scalar   s);
```

## 1447 Parameters

1448           t (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1449                       scalar.

1450           s (IN) The GraphBLAS scalar to be duplicated.

## 1451 Return Values

1452           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1453                       blocking mode, this indicates that the API checks for the input  
1454                       arguments passed successfully. Either way, output scalar  $t$  is ready  
1455                       to be used in the next method of the sequence.

1456           GrB\_PANIC Unknown internal error.

1457           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1458                       GraphBLAS objects (input or output) is in an invalid state caused  
1459                       by a previous execution error. Call `GrB_error()` to access any error  
1460                       messages generated by the implementation.

1461           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1462 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS scalar,  $s$ , has not been initialized by a call to  
1463                       `Scalar_new` or `Scalar_dup`.

1464           GrB\_NULL\_POINTER The  $t$  pointer is NULL.

## 1465 Description

1466 Creates a new scalar  $t$  of domain  $\mathbf{D}(s)$  and contents  $\mathbf{L}(s)$ . The method returns a handle to the new  
1467 scalar in  $t$ .

1468 It is not an error to call this method more than once with the same output variable; however, the  
1469 handle to the previously created object will be overwritten.

### 1470 4.2.3.3 Scalar\_clear: Clear/remove a stored value from a scalar

1471 Removes the stored value from a scalar.

## 1472 C Syntax

1473           GrB\_Info GrB\_Scalar\_clear(GrB\_Scalar s);

## 1474 Parameters

1475            $s$  (INOUT) An existing GraphBLAS scalar to clear.

## 1476 Return Values

1477           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1478                       blocking mode, this indicates that the API checks for the input

1479 arguments passed successfully. Either way, output scalar `s` is ready  
1480 to be used in the next method of the sequence.

1481 `GrB_PANIC` Unknown internal error.

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

1486 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1487 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS scalar, `s`, has not been initialized by a call to  
1488 `Scalar_new` or `Scalar_dup`.

## 1489 Description

1490 Removes the stored value from an existing scalar. After the call, `L(s)` is empty. The size of the  
1491 scalar does not change.

### 1492 4.2.3.4 `Scalar_nvals`: Number of stored elements in a scalar

1493 Retrieve the number of stored elements in a scalar (either zero or one).

## 1494 C Syntax

```
1495 GrB_Info GrB_Scalar_nvals(GrB_Index      *nvals,  
1496                           const GrB_Scalar s);
```

## 1497 Parameters

1498 `nvals` (OUT) On successful return, this is set to the number of stored elements in the  
1499 scalar (zero or one).

1500 `s` (IN) An existing GraphBLAS scalar being queried.

## 1501 Return Values

1502 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-  
1503 cessfully and the value of `nvals` has been set.

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

## 1513 Description

1514 Return *nvals(s)* in *nvals*. This is the number of stored elements in scalar *s*, which is the size of  
 1515 *L(s)*, and can only be either zero or one (see Section 3.5.1).

### 1516 4.2.3.5 Scalar\_setElement: Set the single element in a scalar

1517 Set the single element of a scalar to a given value.

## 1518 C Syntax

```
1519           GrB_Info GrB_Scalar_setElement(GrB_Scalar    s,
1520                                           <type>       val);
```

## 1521 Parameters

1522           *s* (INOUT) An existing GraphBLAS scalar for which the element is to be assigned.

1523           *val* (IN) Scalar value to assign. The type must be compatible with the domain of *s*.

## 1524 Return Values

1525           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 1526           blocking mode, this indicates that the compatibility tests on in-  
 1527           dex/dimensions and domains for the input arguments passed suc-  
 1528           cessfully. Either way, the output scalar *s* is ready to be used in the  
 1529           next method of the sequence.

1530           GrB\_PANIC Unknown internal error.

1531           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 1532           GraphBLAS objects (input or output) is in an invalid state caused

1533 by a previous execution error. Call `GrB_error()` to access any error  
1534 messages generated by the implementation.

1535 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1536 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS scalar, `s`, has not been initialized by a call to  
1537 `Scalar_new` or `Scalar_dup`.

1538 **GrB\_DOMAIN\_MISMATCH** The domains of `s` and `val` are incompatible.

## 1539 Description

1540 First, `val` and output GraphBLAS scalar are tested for domain compatibility as follows: **D**(`val`) must  
1541 be compatible with **D**(`s`). Two domains are compatible with each other if values from one domain  
1542 can be cast to values in the other domain as per the rules of the C language. In particular, domains  
1543 from Table 3.2 are all compatible with each other. A domain from a user-defined type is only com-  
1544 patible with itself. If any compatibility rule above is violated, execution of `GrB_Scalar_setElement`  
1545 ends and the domain mismatch error listed above is returned.

1546 We are now ready to carry out the assignment `val`; that is:

$$1547 \quad s(0) = val$$

1548 If `s` already had a stored value, it will be overwritten; otherwise, the new value is stored in `s`.

1549 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents  
1550 of `s` is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with  
1551 return value `GrB_SUCCESS` and the new content of scalar `s` is as defined above but may not be  
1552 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 1553 4.2.3.6 `Scalar_extractElement`: Extract a single element from a scalar.

1554 Assign a non-opaque scalar with the value of the element stored in a GraphBLAS scalar.

## 1555 C Syntax

```
1556      GrB_Info GrB_Scalar_extractElement(<type>          *val,  
1557                                         const GrB_Scalar s);
```

## 1558 Parameters

1559 `val` (INOUT) Pointer to a non-opaque scalar of type that is compatible with the domain  
1560 of scalar `s`. On successful return, `val` holds the result of the operation, and any  
1561 previous value in `val` is overwritten.

1562 `s` (IN) The GraphBLAS scalar from which an element is extracted.

## 1563 Return Values

1564           **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
1565                           cessfully. This indicates that the compatibility tests on dimensions  
1566                           and domains for the input arguments passed successfully, and the  
1567                           output scalar, `val`, has been computed and is ready to be used in  
1568                           the next method of the sequence.

1569           **GrB\_PANIC** Unknown internal error.

1570           **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1571                           GraphBLAS objects (input or output) is in an invalid state caused  
1572                           by a previous execution error. Call `GrB_error()` to access any error  
1573                           messages generated by the implementation.

1574           **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1575 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS scalar, `s`, has not been initialized by a call to  
1576                           `Scalar_new` or `Scalar_dup`.

1577           **GrB\_NULL\_POINTER** `val` pointer is NULL.

1578           **GrB\_DOMAIN\_MISMATCH** The domains of the scalar or scalar are incompatible.

1579           **GrB\_NO\_VALUE** There is no stored value in the scalar.

## 1580 Description

1581 First, `val` and input GraphBLAS scalar are tested for domain compatibility as follows: **D(val)**  
1582 must be compatible with **D(s)**. Two domains are compatible with each other if values from  
1583 one domain can be cast to values in the other domain as per the rules of the C language. In  
1584 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
1585 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
1586 `GrB_Scalar_extractElement` ends and the domain mismatch error listed above is returned.

1587 Then, if no value is currently stored in the GraphBLAS scalar, the method returns **GrB\_NO\_VALUE**  
1588 and `val` remains unchanged.

1589 Finally the extract into the output argument, `val` can be performed; that is:

1590   
$$\text{val} = \text{s}(0)$$

1591 In both **GrB\_BLOCKING** mode **GrB\_NONBLOCKING** mode if the method exits with return value  
1592 **GrB\_SUCCESS**, the new contents of `val` are as defined above.



## 1593 4.2.4 Vector methods

### 1594 4.2.4.1 Vector\_new: Construct new vector

1595 Creates a new vector with specified domain and size.

## 1596 C Syntax

```
1597         GrB_Info GrB_Vector_new(GrB_Vector *v,  
1598                                GrB_Type    d,  
1599                                GrB_Index   nsize);
```

## 1600 Parameters

1601 v (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1602 vector.

1603 d (IN) The type corresponding to the domain of the vector being created. Can be  
1604 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
1605 GraphBLAS type.

1606 nsize (IN) The size of the vector being created.

## 1607 Return Values

1608 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1609 blocking mode, this indicates that the API checks for the input  
1610 arguments passed successfully. Either way, output vector v is ready  
1611 to be used in the next method of the sequence.

1612 GrB\_PANIC Unknown internal error.

1613 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1614 GraphBLAS objects (input or output) is in an invalid state caused  
1615 by a previous execution error. Call GrB\_error() to access any error  
1616 messages generated by the implementation.

1617 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1618 GrB\_UNINITIALIZED\_OBJECT The GrB\_Type object has not been initialized by a call to GrB\_Type\_new  
1619 (needed for user-defined types).

1620 GrB\_NULL\_POINTER The v pointer is NULL.

1621 GrB\_INVALID\_VALUE nsize is zero or outside the range of the type GrB\_Index.

## 1622 Description

1623 Creates a new vector  $\mathbf{v}$  of domain  $\mathbf{D(d)}$ , size  $\mathbf{nsize}$ , and empty  $\mathbf{L(v)}$ . The method returns a handle  
1624 to the new vector in  $\mathbf{v}$ .

1625 It is not an error to call this method more than once on the same variable; however, the handle to  
1626 the previously created object will be overwritten.

### 1627 4.2.4.2 Vector\_dup: Construct a copy of a GraphBLAS vector

1628 Creates a new vector with the same domain, size, and contents as another vector.

## 1629 C Syntax

```
1630         GrB_Info GrB_Vector_dup(GrB_Vector      *w,  
1631                                const GrB_Vector  u);
```

## 1632 Parameters

1633  $\mathbf{w}$  (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1634 vector.

1635  $\mathbf{u}$  (IN) The GraphBLAS vector to be duplicated.

## 1636 Return Values

1637  $\mathbf{GrB\_SUCCESS}$  In blocking mode, the operation completed successfully. In non-  
1638 blocking mode, this indicates that the API checks for the input  
1639 arguments passed successfully. Either way, output vector  $\mathbf{w}$  is ready  
1640 to be used in the next method of the sequence.

1641  $\mathbf{GrB\_PANIC}$  Unknown internal error.

1642  $\mathbf{GrB\_INVALID\_OBJECT}$  This is returned in any execution mode whenever one of the opaque  
1643 GraphBLAS objects (input or output) is in an invalid state caused  
1644 by a previous execution error. Call  $\mathbf{GrB\_error()}$  to access any error  
1645 messages generated by the implementation.

1646  $\mathbf{GrB\_OUT\_OF\_MEMORY}$  Not enough memory available for operation.

1647  $\mathbf{GrB\_UNINITIALIZED\_OBJECT}$  The GraphBLAS vector,  $\mathbf{u}$ , has not been initialized by a call to  
1648  $\mathbf{Vector\_new}$  or  $\mathbf{Vector\_dup}$ .

1649  $\mathbf{GrB\_NULL\_POINTER}$  The  $\mathbf{w}$  pointer is  $\mathbf{NULL}$ .

## 1650 Description

1651 Creates a new vector  $\mathbf{w}$  of domain  $\mathbf{D}(\mathbf{u})$ , size  $\mathbf{size}(\mathbf{u})$ , and contents  $\mathbf{L}(\mathbf{u})$ . The method returns a  
1652 handle to the new vector in  $\mathbf{w}$ .

1653 It is not an error to call this method more than once on the same variable; however, the handle to  
1654 the previously created object will be overwritten.

### 1655 4.2.4.3 Vector\_resize: Resize a vector

1656 Changes the size of an existing vector.

## 1657 C Syntax

```
1658         GrB_Info GrB_Vector_resize(GrB_Vector  w,  
1659                                   GrB_Index   nsize);
```

## 1660 Parameters

1661  $\mathbf{w}$  (INOUT) An existing Vector object that is being resized.

1662  $\mathbf{nsize}$  (IN) The new size of the vector. It can be smaller or larger than the current size.

## 1663 Return Values

1664  $\mathbf{GrB\_SUCCESS}$  In blocking mode, the operation completed successfully. In non-  
1665 blocking mode, this indicates that the API checks for the input  
1666 arguments passed successfully. Either way, output vector  $\mathbf{w}$  is ready  
1667 to be used in the next method of the sequence.

1668  $\mathbf{GrB\_PANIC}$  Unknown internal error.

1669  $\mathbf{GrB\_INVALID\_OBJECT}$  This is returned in any execution mode whenever one of the opaque  
1670 GraphBLAS objects (input or output) is in an invalid state caused  
1671 by a previous execution error. Call  $\mathbf{GrB\_error()}$  to access any error  
1672 messages generated by the implementation.

1673  $\mathbf{GrB\_OUT\_OF\_MEMORY}$  Not enough memory available for operation.

1674  $\mathbf{GrB\_NULL\_POINTER}$  The  $\mathbf{w}$  pointer is  $\mathbf{NULL}$ .

1675  $\mathbf{GrB\_INVALID\_VALUE}$   $\mathbf{nsize}$  is zero or outside the range of the type  $\mathbf{GrB\_Index}$ .

## 1676 Description

1677 Changes the size of  $w$  to  $nsize$ . The domain  $\mathbf{D}(w)$  of vector  $w$  remains the same. The contents  $\mathbf{L}(w)$   
1678 are modified as described below.

1679 Let  $w = \langle \mathbf{D}(w), N, \mathbf{L}(w) \rangle$  when the method is called. When the method returns,  $w = \langle \mathbf{D}(w), nsize, \mathbf{L}'(w) \rangle$   
1680 where  $\mathbf{L}'(w) = \{(i, w_i) : (i, w_i) \in \mathbf{L}(w) \wedge (i < nsize)\}$ . That is, all elements of  $w$  with index greater  
1681 than or equal to the new vector size ( $nsize$ ) are dropped.

### 1682 4.2.4.4 Vector\_clear: Clear a vector

1683 Removes all the elements (tuples) from a vector.

## 1684 C Syntax

```
1685      GrB_Info GrB_Vector_clear(GrB_Vector v);
```

## 1686 Parameters

1687  $v$  (INOUT) An existing GraphBLAS vector to clear.

## 1688 Return Values

1689 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1690 blocking mode, this indicates that the API checks for the input  
1691 arguments passed successfully. Either way, output vector  $v$  is ready  
1692 to be used in the next method of the sequence.

1693 **GrB\_PANIC** Unknown internal error.

1694 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1695 GraphBLAS objects (input or output) is in an invalid state caused  
1696 by a previous execution error. Call `GrB_error()` to access any error  
1697 messages generated by the implementation.

1698 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1699 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector,  $v$ , has not been initialized by a call to  
1700 `Vector_new` or `Vector_dup`.

## 1701 Description

1702 Removes all elements (tuples) from an existing vector. After the call to `GrB_Vector_clear(v)`,  
1703  $\mathbf{L}(v) = \emptyset$ . The size of the vector does not change.

#### 1704 **4.2.4.5 Vector\_size: Size of a vector**

1705 Retrieve the size of a vector.

#### 1706 **C Syntax**

```
1707         GrB_Info GrB_Vector_size(GrB_Index          *nsize,  
1708                                const GrB_Vector  v);
```

#### 1709 **Parameters**

1710 nsize (OUT) On successful return, is set to the size of the vector.

1711 v (IN) An existing GraphBLAS vector being queried.

#### 1712 **Return Values**

1713 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
1714 cessfully and the value of nsize has been set.

1715 GrB\_PANIC Unknown internal error.

1716 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1717 GraphBLAS objects (input or output) is in an invalid state caused  
1718 by a previous execution error. Call GrB\_error() to access any error  
1719 messages generated by the implementation.

1720 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, v, has not been initialized by a call to  
1721 Vector\_new or Vector\_dup.

1722 GrB\_NULL\_POINTER nsize pointer is NULL.

#### 1723 **Description**

1724 Return **size**(v) in nsize.

#### 1725 **4.2.4.6 Vector\_nvals: Number of stored elements in a vector**

1726 Retrieve the number of stored elements (tuples) in a vector.

#### 1727 **C Syntax**

```
1728         GrB_Info GrB_Vector_nvals(GrB_Index          *nvals,  
1729                                const GrB_Vector  v);
```

## 1730 Parameters

1731            **nvals** (OUT) On successful return, this is set to the number of stored elements (tuples)  
1732            in the vector.

1733            **v** (IN) An existing GraphBLAS vector being queried.

## 1734 Return Values

1735            **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
1736            cessfully and the value of **nvals** has been set.

1737            **GrB\_PANIC** Unknown internal error.

1738            **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1739            GraphBLAS objects (input or output) is in an invalid state caused  
1740            by a previous execution error. Call **GrB\_error()** to access any error  
1741            messages generated by the implementation.

1742            **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

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** The **nvals** pointer is **NULL**.

## 1746 Description

1747            Return **nvals(v)** in **nvals**. This is the number of stored elements in vector **v**, which is the size of  
1748            **L(v)** (see Section 3.5.2).

### 1749 4.2.4.7 Vector\_build: Store elements from tuples into a vector

## 1750 C Syntax

```
1751            GrB_Info GrB_Vector_build(GrB_Vector            w,  
1752                                        const GrB_Index        *indices,  
1753                                        const <type>           *values,  
1754                                        GrB_Index                n,  
1755                                        const GrB_BinaryOp       dup);
```

## 1756 Parameters

1757            **w** (INOUT) An existing Vector object to store the result.

1758            **indices** (IN) Pointer to an array of indices.

1759        **values** (IN) Pointer to an array of scalars of a type that is compatible with the domain of  
 1760        vector **w**.

1761        **n** (IN) The number of entries contained in each array (the same for indices and values).

1762        **dup** (IN) An associative and commutative binary operator to apply when duplicate  
 1763        values for the same location are present in the input arrays. All three domains of  
 1764        **dup** must be the same; hence  $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$ . If **dup** is **GrB\_NULL**,  
 1765        then duplicate locations will result in an error.

## 1766    **Return Values**

1767        **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 1768        blocking mode, this indicates that the API checks for the input  
 1769        arguments passed successfully. Either way, output vector **w** is  
 1770        ready to be used in the next method of the sequence.

1771        **GrB\_PANIC** Unknown internal error.

1772        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
 1773        opaque GraphBLAS objects (input or output) is in an invalid  
 1774        state caused by a previous execution error. Call **GrB\_error()** to  
 1775        access any error messages generated by the implementation.

1776        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1777        **GrB\_UNINITIALIZED\_OBJECT** Either **w** has not been initialized by a call to **GrB\_Vector\_new**  
 1778        or by **GrB\_Vector\_dup**, or **dup** has not been initialized by a call  
 1779        to **GrB\_BinaryOp\_new**.

1780        **GrB\_NULL\_POINTER** indices or values pointer is **NULL**.

1781        **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in indices is outside the allowed range for **w**.

1782        **GrB\_DOMAIN\_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are  
 1783        not all the same, or the domains of **values** and **w** are incompatible  
 1784        with each other or  $D_{dup}$ .

1785        **GrB\_OUTPUT\_NOT\_EMPTY** Output vector **w** already contains valid tuples (elements). In  
 1786        other words, **GrB\_Vector\_nvals(C)** returns a positive value.

1787        **GrB\_INVALID\_VALUE** indices contains a duplicate location and **dup** is **GrB\_NULL**.

## 1788    **Description**

1789    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  
 1790    from **w** in its domain; otherwise,  $\tilde{\mathbf{w}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \emptyset \rangle$ .

1791 Each tuple  $\{\text{indices}[k], \text{values}[k]\}$ , where  $0 \leq k < n$ , is a contribution to the output in the form of

$$1792 \quad \tilde{\mathbf{w}}(\text{indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } dup \neq \text{GrB\_NULL} \\ (\mathbf{D}(\mathbf{w})) \text{values}[k] & \text{otherwise.} \end{cases}$$

1793 If multiple values for the same location are present in the input arrays and `dup` is not `GrB_NULL`,  
1794 `dup` is used to reduce the values before assignment into  $\tilde{\mathbf{w}}$  as follows:

$$1795 \quad \tilde{\mathbf{w}}_i = \bigoplus_{k: \text{indices}[k]=i} (D_{dup}) \text{values}[k],$$

1796 where  $\oplus$  is the `dup` binary operator. Finally, the resulting  $\tilde{\mathbf{w}}$  is copied into  $\mathbf{w}$  via typecasting its  
1797 values to  $\mathbf{D}(\mathbf{w})$  if necessary. If  $\oplus$  is not associative or not commutative, the result is undefined.

1798 The nonopaque input arrays, `indices` and `values`, must be at least as large as `n`.

1799 It is an error to call this function on an output object with existing elements. In other words,  
1800 `GrB_Vector_nvals(w)` should evaluate to zero prior to calling this function.

1801 After `GrB_Vector_build` returns, it is safe for a programmer to modify or delete the arrays `indices`  
1802 or `values`.

#### 1803 4.2.4.8 Vector\_setElement: Set a single element in a vector

1804 Set one element of a vector to a given value.

### 1805 C Syntax

```
1806 // scalar value
1807 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1808                               <type>          val,
1809                               GrB_Index        index);
1810
1811 // GraphBLAS scalar
1812 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1813                               const GrB_Scalar s,
1814                               GrB_Index        index);
```

### 1815 Parameters

1816 `w` (INOUT) An existing GraphBLAS vector for which an element is to be assigned.

1817 `val` or `s` (IN) Scalar assign. Its domain (type) must be compatible with the domain of `w`.

1818 `index` (IN) The location of the element to be assigned.



## 1819 Return Values

1820           **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1821                           blocking mode, this indicates that the compatibility tests on in-  
1822                           dex/dimensions and domains for the input arguments passed suc-  
1823                           cessfully. Either way, the output vector **w** is ready to be used in  
1824                           the next method of the sequence.

1825           **GrB\_PANIC** Unknown internal error.

1826           **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1827                           GraphBLAS objects (input or output) is in an invalid state caused  
1828                           by a previous execution error. Call **GrB\_error()** to access any error  
1829                           messages generated by the implementation.

1830           **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1831 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **w**, or GraphBLAS scalar, **s**, has not been  
1832                           initialized by a call to a respective constructor.

1833           **GrB\_INVALID\_INDEX** **index** specifies a location that is outside the dimensions of **w**.

1834           **GrB\_DOMAIN\_MISMATCH** The domains of the vector and the scalar are incompatible.

## 1835 Description

1836 First, the scalar and output vector are tested for domain compatibility as follows: **D(val)** or **D(s)**  
1837 must be compatible with **D(w)**. Two domains are compatible with each other if values from  
1838 one domain can be cast to values in the other domain as per the rules of the C language. In  
1839 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
1840 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
1841 **GrB\_Vector\_setElement** ends and the domain mismatch error listed above is returned.

1842 Then, the **index** parameter is checked for a valid value where the following condition must hold:

$$1843 \qquad 0 \leq \text{index} < \text{size}(\mathbf{w})$$

1844 If this condition is violated, execution of **GrB\_Vector\_setElement** ends and the invalid index error  
1845 listed above is returned.

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

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

1846 In the case of a transparent scalar or if **L(s)** is not empty, then a value will be stored at the  
1847 specified location in **w**, overwriting any value that may have been stored there before. In the case  
1848 of a GraphBLAS scalar, if **L(s)** is empty, then any value stored at the specified location in **w** will  
1849 be removed.

1850 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new contents  
 1851 of **w** is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method exits with  
 1852 return value GrB\_SUCCESS and the new contents of vector **w** is as defined above but may not be  
 1853 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 1854 4.2.4.9 Vector\_removeElement: Remove an element from a vector

1855 Remove (annihilate) one stored element from a vector.

#### 1856 C Syntax

```
1857         GrB_Info GrB_Vector_removeElement(GrB_Vector  w,
1858                                           GrB_Index   index);
```

#### 1859 Parameters

1860 **w** (INOUT) An existing GraphBLAS vector from which an element is to be removed.

1861 **index** (IN) The location of the element to be removed.

#### 1862 Return Values

1863 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 1864 blocking mode, this indicates that the compatibility tests on in-  
 1865 dex/dimensions and domains for the input arguments passed suc-  
 1866 cessfully. Either way, the output vector **w** is ready to be used in  
 1867 the next method of the sequence.

1868 **GrB\_PANIC** Unknown internal error.

1869 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 1870 GraphBLAS objects (input or output) is in an invalid state caused  
 1871 by a previous execution error. Call **GrB\_error()** to access any error  
 1872 messages generated by the implementation.

1873 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1874 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **w**, has not been initialized by a call to  
 1875 **Vector\_new** or **Vector\_dup**.

1876 **GrB\_INVALID\_INDEX** **index** specifies a location that is outside the dimensions of **w**.

## 1877 Description

1878 First, the `index` parameter is checked for a valid value where the following condition must hold:

$$1879 \quad 0 \leq \text{index} < \text{size}(\mathbf{w})$$

1880 If this condition is violated, execution of `GrB_Vector_removeElement` ends and the invalid index  
1881 error listed above is returned.

1882 We are now ready to carry out the removal of a value that may be stored at the location specified  
1883 by `index`. If a value does not exist at the specified location in  $\mathbf{w}$ , no error is reported and the  
1884 operation has no effect on the state of  $\mathbf{w}$ . In either case, the following will be true on return from  
1885 the method: `index`  $\notin$  `ind(w)`.

1886 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents  
1887 of  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with  
1888 return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but may not be  
1889 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 1890 4.2.4.10 Vector\_extractElement: Extract a single element from a vector.

1891 Extract one element of a vector into a scalar.

## 1892 C Syntax

```
1893 // scalar value
1894 GrB_Info GrB_Vector_extractElement(<type>          *val,
1895                                   const GrB_Vector u,
1896                                   GrB_Index         index);
1897
1898 // GraphBLAS scalar
1899 GrB_Info GrB_Vector_extractElement(GrB_Scalar      s,
1900                                   const GrB_Vector u,
1901                                   GrB_Index         index);
```

## 1902 Parameters

1903 `val` or `s` (INOUT) An existing scalar of whose domain is compatible with the domain of vector  
1904 `u`. On successful return, this scalar holds the result of the extract. Any previous  
1905 value stored in `val` or `s` is overwritten.

1906 `u` (IN) The GraphBLAS vector from which an element is extracted.

1907 `index` (IN) The location in `u` to extract.

## 1908 Return Values

1909           GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
1910                       cessfully. This indicates that the compatibility tests on dimensions  
1911                       and domains for the input arguments passed successfully, and the  
1912                       output scalar, **val** or **s**, has been computed and is ready to be used  
1913                       in the next method of the sequence.

1914           GrB\_NO\_VALUE When using the transparent scalar, **val**, this is returned when there  
1915                       is no stored value at specified location.

1916           GrB\_PANIC Unknown internal error.

1917           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1918                       GraphBLAS objects (input or output) is in an invalid state caused  
1919                       by a previous execution error. Call **GrB\_error()** to access any error  
1920                       messages generated by the implementation.

1921           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1922 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, **u**, or scalar, **s**, has not been initialized by  
1923                       a call to a corresponding constructor.

1924           GrB\_NULL\_POINTER **val** pointer is NULL.

1925           GrB\_INVALID\_INDEX **index** specifies a location that is outside the dimensions of **w**.

1926           GrB\_DOMAIN\_MISMATCH The domains of the vector and scalar are incompatible.

## 1927 Description

1928 First, the scalar and input vector are tested for domain compatibility as follows: **D(val)** or **D(s)**  
1929 must be compatible with **D(u)**. Two domains are compatible with each other if values from  
1930 one domain can be cast to values in the other domain as per the rules of the C language. In  
1931 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
1932 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
1933 **GrB\_Vector\_extractElement** ends and the domain mismatch error listed above is returned.

1934 Then, the **index** parameter is checked for a valid value where the following condition must hold:

$$1935 \qquad 0 \leq \text{index} < \text{size}(\mathbf{u})$$

1936 If this condition is violated, execution of **GrB\_Vector\_extractElement** ends and the invalid index  
1937 error listed above is returned.

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

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

1938 If  $\text{index} \in \mathbf{ind}(u)$ , then the corresponding value from  $u$  is copied into  $s$  or  $val$  with casting as  
 1939 necessary. If  $\text{index} \notin \mathbf{ind}(u)$ , then one of the follow occurs depending on output scalar type:

- 1940 • The GraphBLAS scalar,  $s$ , is cleared and `GrB_SUCCESS` is returned.
- 1941 • The non-opaque scalar,  $val$ , is unchanged, and `GrB_NO_VALUE` is returned.

1942 When using the non-opaque scalar variant ( $val$ ) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`  
 1943 mode, the new contents of  $val$  are as defined above if the method exits with return value `GrB_SUCCESS`  
 1944 or `GrB_NO_VALUE`.

1945 When using the GraphBLAS scalar variant ( $s$ ) with a `GrB_SUCCESS` return value, the method  
 1946 exits and the new contents of  $s$  is as defined above and fully computed in `GrB_BLOCKING` mode.  
 1947 In `GrB_NONBLOCKING` mode, the new contents of  $s$  is as defined above but may not be fully  
 1948 computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 1949 4.2.4.11 Vector\_extractTuples: Extract tuples from a vector

1950 Extract the contents of a GraphBLAS vector into non-opaque data structures.

### 1951 C Syntax

```

1952      GrB_Info GrB_Vector_extractTuples(GrB_Index      *indices,
1953                                     <type>          *values,
1954                                     GrB_Index        *n,
1955                                     const GrB_Vector  v);
1956
```

1957 **indices** (OUT) Pointer to an array of indices that is large enough to hold all of the stored  
 1958 values' indices.

1959 **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of  
 1960 the stored values whose type is compatible with  $\mathbf{D}(v)$ .

1961 **n** (INOUT) Pointer to a value indicating (on input) the number of elements the  
 1962 values and indices arrays can hold. Upon return, it will contain the number of  
 1963 values written to the arrays.

1964 **v** (IN) An existing GraphBLAS vector.

### 1965 Return Values

1966 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
 1967 cessfully. This indicates that the compatibility tests on the input  
 1968 argument passed successfully, and the output arrays, **indices** and  
 1969 **values**, have been computed.

1970                   GrB\_PANIC Unknown internal error.

1971           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1972                   GraphBLAS objects (input or output) is in an invalid state caused  
1973                   by a previous execution error. Call GrB\_error() to access any error  
1974                   messages generated by the implementation.

1975           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1976           GrB\_INSUFFICIENT\_SPACE Not enough space in `indices` and `values` (as indicated by the `n` pa-  
1977                   rameter) to hold all of the tuples that will be extracted.

1978 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, `v`, has not been initialized by a call to  
1979                   Vector\_new or Vector\_dup.

1980           GrB\_NULL\_POINTER `indices`, `values`, or `n` pointer is NULL.

1981           GrB\_DOMAIN\_MISMATCH The domains of the `v` vector or `values` array are incompatible with  
1982                   one another.

## 1983 Description

1984 This method will extract all the tuples from the GraphBLAS vector `v`. The values associated  
1985 with those tuples are placed in the `values` array and the indices are placed in the `indices` array.  
1986 Both `indices` and `values` must be pre-allocated by the user to have enough space to hold at least  
1987 GrB\_Vector\_nvals(`v`) elements before calling this function.

1988 Upon return of this function, `n` will be set to the number of values (and indices) copied. Also, the  
1989 entries of `indices` are unique, but not necessarily sorted. Each tuple  $(i, v_i)$  in `v` is unzipped and  
1990 copied into a distinct  $k$ th location in output vectors:

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

1991 where  $0 \leq k < \text{GrB\_Vector\_nvals}(v)$ . No gaps in output vectors are allowed; that is, if `indices`[ $k$ ]  
1992 and `values`[ $k$ ] exist upon return, so does `indices`[ $j$ ] and `values`[ $j$ ] for all  $j$  such that  $0 \leq j < k$ .

1993 Note that if the value in `n` on input is less than the number of values contained in the vector `v`,  
1994 then a GrB\_INSUFFICIENT\_SPACE error is returned because it is undefined which subset of values  
1995 would be extracted otherwise.

1996 In both GrB\_BLOCKING mode GrB\_NONBLOCKING mode if the method exits with return value  
1997 GrB\_SUCCESS, the new contents of the arrays `indices` and `values` are as defined above.

## 1998 4.2.5 Matrix methods

### 1999 4.2.5.1 Matrix\_new: Construct new matrix

2000 Creates a new matrix with specified domain and dimensions.

## 2001 C Syntax

```
2002         GrB_Info GrB_Matrix_new(GrB_Matrix *A,  
2003                                 GrB_Type      d,  
2004                                 GrB_Index     nrows,  
2005                                 GrB_Index     ncols);
```

## 2006 Parameters

2007 **A** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2008 matrix.

2009 **d** (IN) The type corresponding to the domain of the matrix being created. Can be  
2010 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
2011 GraphBLAS type.

2012 **nrows** (IN) The number of rows of the matrix being created.

2013 **ncols** (IN) The number of columns of the matrix being created.

## 2014 Return Values

2015 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2016 blocking mode, this indicates that the API checks for the input ar-  
2017 guments passed successfully. Either way, output matrix **A** is ready  
2018 to be used in the next method of the sequence.

2019 **GrB\_PANIC** Unknown internal error.

2020 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2021 GraphBLAS objects (input or output) is in an invalid state caused  
2022 by a previous execution error. Call **GrB\_error()** to access any error  
2023 messages generated by the implementation.

2024 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2025 **GrB\_UNINITIALIZED\_OBJECT** The **GrB\_Type** object has not been initialized by a call to **GrB\_Type\_new**  
2026 (needed for user-defined types).

2027 **GrB\_NULL\_POINTER** The **A** pointer is **NULL**.

2028 **GrB\_INVALID\_VALUE** **nrows** or **ncols** is zero or outside the range of the type **GrB\_Index**.

## 2029 Description

2030 Creates a new matrix **A** of domain **D**(**d**), size **nrows**  $\times$  **ncols**, and empty **L**(**A**). The method returns  
2031 a handle to the new matrix in **A**.

2032 It is not an error to call this method more than once on the same variable; however, the handle to  
2033 the previously created object will be overwritten.

#### 2034 4.2.5.2 Matrix\_dup: Construct a copy of a GraphBLAS matrix

2035 Creates a new matrix with the same domain, dimensions, and contents as another matrix.

#### 2036 C Syntax

```
2037         GrB_Info GrB_Matrix_dup(GrB_Matrix      *C,  
2038                                const GrB_Matrix  A);
```

#### 2039 Parameters

2040 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2041 matrix.

2042 A (IN) The GraphBLAS matrix to be duplicated.

#### 2043 Return Values

2044 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2045 blocking mode, this indicates that the API checks for the input  
2046 arguments passed successfully. Either way, output matrix C is ready  
2047 to be used in the next method of the sequence.

2048 GrB\_PANIC Unknown internal error.

2049 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2050 GraphBLAS objects (input or output) is in an invalid state caused  
2051 by a previous execution error. Call GrB\_error() to access any error  
2052 messages generated by the implementation.

2053 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2054 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2055 any matrix constructor.

2056 GrB\_NULL\_POINTER The C pointer is NULL.

#### 2057 Description

2058 Creates a new matrix C of domain D(A), size nrows(A) × ncols(A), and contents L(A). It returns  
2059 a handle to it in C.



2060 It is not an error to call this method more than once on the same variable; however, the handle to  
2061 the previously created object will be overwritten.

#### 2062 4.2.5.3 Matrix\_diag: Construct a diagonal GraphBLAS matrix

2063 Creates a new matrix with the same domain and contents as a GrB\_Vector, and square dimensions  
2064 appropriate for placing the contents of the vector along the specified diagonal of the matrix.

#### 2065 C Syntax

```
2066         GrB_Info GrB_Matrix_diag(GrB_Matrix      *C,  
2067                                 const GrB_Vector  v,  
2068                                 int64_t           k);
```

#### 2069 Parameters

2070 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2071 matrix. The matrix is square with each dimension equal to **size(v) + |k|**.

2072 v (IN) The GraphBLAS vector whose contents will be copied to the diagonal of the  
2073 matrix.

2074 k (IN) The diagonal to which the vector is assigned. k = 0 represents the main  
2075 diagonal, k > 0 is above the main diagonal, and k < 0 is below.

#### 2076 Return Values

2077 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2078 blocking mode, this indicates that the API checks for the input  
2079 arguments passed successfully. Either way, output matrix C is ready  
2080 to be used in the next method of the sequence.

2081 GrB\_PANIC Unknown internal error.

2082 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2083 GraphBLAS objects (input or output) is in an invalid state caused  
2084 by a previous execution error. Call GrB\_error() to access any error  
2085 messages generated by the implementation.

2086 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

2087 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, v, has not been initialized by a call to  
2088 Vector\_new or Vector\_dup.

2089 GrB\_NULL\_POINTER The C pointer is NULL.

## 2090 Description

2091 Creates a new matrix **C** of domain **D(v)**, size  $(\mathbf{size}(\mathbf{v}) + |k|) \times (\mathbf{size}(\mathbf{v}) + |k|)$ , and contents

$$\begin{aligned} 2092 \quad \mathbf{L}(\mathbf{C}) &= \{(i, i + k, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k \geq 0 \text{ or} \\ 2093 \quad \mathbf{L}(\mathbf{C}) &= \{(i - k, i, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k < 0. \end{aligned}$$

2094 It returns a handle to it in **C**. It is not an error to call this method more than once on the same  
2095 variable; however, the handle to the previously created object will be overwritten.

## 2096 4.2.5.4 Matrix\_resize: Resize a matrix

2097 Changes the dimensions of an existing matrix.

## 2098 C Syntax

```
2099         GrB_Info GrB_Matrix_resize(GrB_Matrix  C,  
2100                                   GrB_Index   nrows,  
2101                                   GrB_Index   ncols);
```

## 2102 Parameters

2103 **C** (INOUT) An existing Matrix object that is being resized.

2104 **nrows** (IN) The new number of rows of the matrix. It can be smaller or larger than the  
2105 current number of rows.

2106 **ncols** (IN) The new number of columns of the matrix. It can be smaller or larger than  
2107 the current number of columns.

## 2108 Return Values

2109 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2110 blocking mode, this indicates that the API checks for the input  
2111 arguments passed successfully. Either way, output matrix **C** is ready  
2112 to be used in the next method of the sequence.

2113 **GrB\_PANIC** Unknown internal error.

2114 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2115 GraphBLAS objects (input or output) is in an invalid state caused  
2116 by a previous execution error. Call **GrB\_error()** to access any error  
2117 messages generated by the implementation.

2118 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2119           GrB\_NULL\_POINTER The C pointer is NULL.

2120           GrB\_INVALID\_VALUE nrows or ncols is zero or outside the range of the type GrB\_Index.

## 2121   **Description**

2122   Changes the number of rows and columns of C to nrows and ncols, respectively. The domain  $\mathbf{D}(\mathbf{C})$   
2123   of matrix C remains the same. The contents  $\mathbf{L}(\mathbf{C})$  are modified as described below.

2124   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  
2125   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  
2126   greater than or equal to ncols are dropped.  
2127

### 2128   **4.2.5.5   Matrix\_clear: Clear a matrix**

2129   Removes all elements (tuples) from a matrix.

## 2130   **C Syntax**

2131           GrB\_Info GrB\_Matrix\_clear(GrB\_Matrix A);

## 2132   **Parameters**

2133           A (IN) An existing GraphBLAS matrix to clear.

## 2134   **Return Values**

2135           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2136                       blocking mode, this indicates that the API checks for the input ar-  
2137                       guments passed successfully. Either way, output matrix A is ready  
2138                       to be used in the next method of the sequence.

2139           GrB\_PANIC Unknown internal error.

2140           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2141                               GraphBLAS objects (input or output) is in an invalid state caused  
2142                               by a previous execution error. Call GrB\_error() to access any error  
2143                               messages generated by the implementation.

2144           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2145           GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2146                               any matrix constructor.

2147 **Description**

2148 Removes all elements (tuples) from an existing matrix. After the call to `GrB_Matrix_clear(A)`,  
2149  $\mathbf{L}(\mathbf{A}) = \emptyset$ . The dimensions of the matrix do not change.

2150 **4.2.5.6 Matrix\_nrows: Number of rows in a matrix**

2151 Retrieve the number of rows in a matrix.

2152 **C Syntax**

```
2153         GrB_Info GrB_Matrix_nrows(GrB_Index      *nrows,  
2154                                   const GrB_Matrix A);
```

2155 **Parameters**

2156 nrows (OUT) On successful return, contains the number of rows in the matrix.

2157 A (IN) An existing GraphBLAS matrix being queried.

2158 **Return Values**

2159 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
2160 cessfully and the value of `nrows` has been set.

2161 GrB\_PANIC Unknown internal error.

2162 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2163 GraphBLAS objects (input or output) is in an invalid state caused  
2164 by a previous execution error. Call `GrB_error()` to access any error  
2165 messages generated by the implementation.

2166 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to  
2167 any matrix constructor.

2168 GrB\_NULL\_POINTER `nrows` pointer is NULL.

2169 **Description**

2170 Return `nrows(A)` in `nrows` (the number of rows).

2171 **4.2.5.7 Matrix\_ncols: Number of columns in a matrix**

2172 Retrieve the number of columns in a matrix.

## 2173 C Syntax

```
2174         GrB_Info GrB_Matrix_ncols(GrB_Index      *ncols,  
2175                                   const GrB_Matrix A);
```

## 2176 Parameters

2177 ncols (OUT) On successful return, contains the number of columns in the matrix.

2178 A (IN) An existing GraphBLAS matrix being queried.

## 2179 Return Values

2180 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
2181 cessfully and the value of ncols has been set.

2182 GrB\_PANIC Unknown internal error.

2183 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2184 GraphBLAS objects (input or output) is in an invalid state caused  
2185 by a previous execution error. Call GrB\_error() to access any error  
2186 messages generated by the implementation.

2187 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2188 any matrix constructor.

2189 GrB\_NULL\_POINTER ncols pointer is NULL.

## 2190 Description

2191 Return **ncols(A)** in ncols (the number of columns).

## 2192 4.2.5.8 Matrix\_nvals: Number of stored elements in a matrix

2193 Retrieve the number of stored elements (tuples) in a matrix.

## 2194 C Syntax

```
2195         GrB_Info GrB_Matrix_nvals(GrB_Index      *nvals,  
2196                                   const GrB_Matrix A);
```

2197 **Parameters**

2198            **nvals** (OUT) On successful return, contains the number of stored elements (tuples) in  
2199            the matrix.

2200            **A** (IN) An existing GraphBLAS matrix being queried.

2201 **Return Values**

2202            **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2203            cessfully and the value of **nvals** has been set.

2204            **GrB\_PANIC** Unknown internal error.

2205            **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2206            GraphBLAS objects (input or output) is in an invalid state caused  
2207            by a previous execution error. Call **GrB\_error()** to access any error  
2208            messages generated by the implementation.

2209            **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2210            **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to  
2211            any matrix constructor.

2212            **GrB\_NULL\_POINTER** The **nvals** pointer is **NULL**.

2213 **Description**

2214            Return **nvals(A)** in **nvals**. This is the number of tuples stored in matrix **A**, which is the size of  
2215            **L(A)** (see Section 3.5.3).

2216 **4.2.5.9 Matrix\_build: Store elements from tuples into a matrix**

2217 **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);
```

2218 **Parameters**

2219            **C** (INOUT) An existing Matrix object to store the result.

2220 **row\_indices** (IN) Pointer to an array of row indices.

2221 **col\_indices** (IN) Pointer to an array of column indices.

2222 **values** (IN) Pointer to an array of scalars of a type that is compatible with the domain of  
 2223 matrix, **C**.

2224 **n** (IN) The number of entries contained in each array (the same for **row\_indices**,  
 2225 **col\_indices**, and **values**).

2226 **dup** (IN) An associative and commutative binary operator to apply when duplicate  
 2227 values for the same location are present in the input arrays. All three domains of  
 2228 **dup** must be the same; hence  $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$ . If **dup** is **GrB\_NULL**,  
 2229 then duplicate locations will result in an error.

## 2230 Return Values

2231 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 2232 blocking mode, this indicates that the API checks for the input  
 2233 arguments passed successfully. Either way, output matrix **C** is  
 2234 ready to be used in the next method of the sequence.

2235 **GrB\_PANIC** Unknown internal error.

2236 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
 2237 opaque GraphBLAS objects (input or output) is in an invalid  
 2238 state caused by a previous execution error. Call **GrB\_error()** to  
 2239 access any error messages generated by the implementation.

2240 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2241 **GrB\_UNINITIALIZED\_OBJECT** Either **C** has not been initialized by a call to any matrix construc-  
 2242 tor, or **dup** has not been initialized by a call to **GrB\_BinaryOp\_new**.

2243 **GrB\_NULL\_POINTER** **row\_indices**, **col\_indices** or **values** pointer is **NULL**.

2244 **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **row\_indices** or **col\_indices** is outside the allowed range  
 2245 for **C**.

2246 **GrB\_DOMAIN\_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are  
 2247 not all the same, or the domains of **values** and **C** are incompatible  
 2248 with each other or  $D_{dup}$ .

2249 **GrB\_OUTPUT\_NOT\_EMPTY** Output matrix **C** already contains valid tuples (elements). In  
 2250 other words, **GrB\_Matrix\_nvals(C)** returns a positive value.

2251 **GrB\_INVALID\_VALUE** indices contains a duplicate location and **dup** is **GrB\_NULL**.

## 2252 Description

2253 If `dup` is not `GrB_NULL`, an internal matrix  $\tilde{\mathbf{C}} = \langle D_{dup}, \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \emptyset \rangle$  is created, which  
 2254 only differs from  $\mathbf{C}$  in its domain; otherwise,  $\tilde{\mathbf{C}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \emptyset \rangle$ .

2255 Each tuple  $\{\text{row\_indices}[k], \text{col\_indices}[k], \text{values}[k]\}$ , where  $0 \leq k < n$ , is a contribution to the  
 2256 output in the form of

$$2257 \quad \tilde{\mathbf{C}}(\text{row\_indices}[k], \text{col\_indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \text{dup} \neq \text{GrB\_NULL} \\ (\mathbf{D}(\mathbf{C})) \text{values}[k] & \text{otherwise.} \end{cases}$$

2258 If multiple values for the same location are present in the input arrays and `dup` is not `GrB_NULL`,  
 2259 `dup` is used to reduce the values before assignment into  $\tilde{\mathbf{C}}$  as follows:

$$2260 \quad \tilde{\mathbf{C}}_{ij} = \bigoplus_{k: \text{row\_indices}[k]=i \wedge \text{col\_indices}[k]=j} (D_{dup}) \text{values}[k],$$

2261 where  $\oplus$  is the `dup` binary operator. Finally, the resulting  $\tilde{\mathbf{C}}$  is copied into  $\mathbf{C}$  via typecasting its  
 2262 values to  $\mathbf{D}(\mathbf{C})$  if necessary. If  $\oplus$  is not associative or not commutative, the result is undefined.

2263 The nonopaque input arrays `row_indices`, `col_indices`, and `values` must be at least as large as `n`.

2264 It is an error to call this function on an output object with existing elements. In other words,  
 2265 `GrB_Matrix_nvals(C)` should evaluate to zero prior to calling this function.

2266 After `GrB_Matrix_build` returns, it is safe for a programmer to modify or delete the arrays `row_indices`,  
 2267 `col_indices`, or `values`.

### 2268 4.2.5.10 Matrix\_setElement: Set a single element in matrix

2269 Set one element of a matrix to a given value.

## 2270 C Syntax

```
2271 // scalar value
2272 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2273                                <type>         val,
2274                                GrB_Index        row_index,
2275                                GrB_Index        col_index);
2276
2277 // GraphBLAS scalar
2278 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2279                                const GrB_Scalar s,
2280                                GrB_Index        row_index,
2281                                GrB_Index        col_index);
```



## 2282 Parameters

2283           **C** (INOUT) An existing GraphBLAS matrix for which an element is to be assigned.  
2284           **val** or **s** (IN) Scalar to assign. Its domain (type) must be compatible with the domain of  
2285           **C**.  
2286           **row\_index** (IN) Row index of element to be assigned  
2287           **col\_index** (IN) Column index of element to be assigned

## 2288 Return Values

2289           **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2290           blocking mode, this indicates that the compatibility tests on in-  
2291           dex/dimensions and domains for the input arguments passed suc-  
2292           cessfully. Either way, the output matrix **C** is ready to be used in  
2293           the next method of the sequence.  
2294           **GrB\_PANIC** Unknown internal error.  
2295           **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2296           GraphBLAS objects (input or output) is in an invalid state caused  
2297           by a previous execution error. Call **GrB\_error()** to access any error  
2298           messages generated by the implementation.  
2299           **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.  
2300           **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, or GraphBLAS scalar, **s**, has not been  
2301           initialized by a call to a respective constructor.  
2302           **GrB\_INVALID\_INDEX** **row\_index** or **col\_index** is outside the allowable range (i.e., not less  
2303           than **nrows(C)** or **ncols(C)**, respectively).  
2304           **GrB\_DOMAIN\_MISMATCH** The domains of the matrix and the scalar are incompatible.

## 2305 Description

2306 First, the scalar and output matrix are tested for domain compatibility as follows: **D(val)** or  
2307 **D(s)** must be compatible with **D(C)**. Two domains are compatible with each other if values from  
2308 one domain can be cast to values in the other domain as per the rules of the C language. In  
2309 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
2310 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
2311 **GrB\_Matrix\_setElement** ends and the domain mismatch error listed above is returned.

2312 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2313 \quad & 0 \leq \text{row\_index} < \text{nrows}(\mathbf{C}), \\ & 0 \leq \text{col\_index} < \text{ncols}(\mathbf{C}) \end{aligned}$$

2314 If either of these conditions is violated, execution of `GrB_Matrix_setElement` ends and the invalid  
 2315 index error listed above is returned.

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

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

2316 In the case of a transparent scalar or if  $\mathbf{L}(s)$  is not empty, then a value will be stored at the  
 2317 specified location in  $C$ , overwriting any value that may have been stored there before. In the case  
 2318 of a GraphBLAS scalar and if  $\mathbf{L}(s)$  is empty, then any value stored at the specified location in  $C$   
 2319 will be removed.

2320 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents  
 2321 of  $C$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with  
 2322 return value `GrB_SUCCESS` and the new content of vector  $C$  is as defined above but may not be  
 2323 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 2324 **4.2.5.11 `Matrix_removeElement`: Remove an element from a matrix**

2325 Remove (annihilate) one stored element from a matrix.

#### 2326 **C Syntax**

```
2327      GrB_Info GrB_Matrix_removeElement(GrB_Matrix  C,
2328                                         GrB_Index   row_index,
2329                                         GrB_Index   col_index);
```

#### 2330 **Parameters**

2331 `C` (INOUT) An existing GraphBLAS matrix from which an element is to be removed.

2332 `row_index` (IN) Row index of element to be removed

2333 `col_index` (IN) Column index of element to be removed

#### 2334 **Return Values**

2335 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
 2336 blocking mode, this indicates that the compatibility tests on in-  
 2337 dex/dimensions and domains for the input arguments passed suc-  
 2338 cessfully. Either way, the output matrix  $C$  is ready to be used in  
 2339 the next method of the sequence.

2340 `GrB_PANIC` Unknown internal error.

2341       GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 2342       GraphBLAS objects (input or output) is in an invalid state caused  
 2343       by a previous execution error. Call GrB\_error() to access any error  
 2344       messages generated by the implementation.

2345       GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2346 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, C, has not been initialized by a call to  
 2347       any matrix constructor.

2348       GrB\_INVALID\_INDEX row\_index or col\_index is outside the allowable range (i.e., not less  
 2349       than nrows(C) or ncols(C), respectively).

## 2350 Description

2351 First, both index parameters are checked for valid values where following conditions must hold:

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

2353 If either of these conditions is violated, execution of GrB\_Matrix\_removeElement ends and the  
 2354 invalid index error listed above is returned.

2355 We are now ready to carry out the removal of a value that may be stored at the location specified by  
 2356 (row\_index, col\_index). If a value does not exist at the specified location in C, no error is reported  
 2357 and the operation has no effect on the state of C. In either case, the following will be true on return  
 2358 from this method: (row\_index, col\_index)  $\notin$  ind(C)

2359 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new contents  
 2360 of C is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method exits with  
 2361 return value GrB\_SUCCESS and the new content of vector C is as defined above but may not be  
 2362 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 2363 4.2.5.12 Matrix\_extractElement: Extract a single element from a matrix

2364 Extract one element of a matrix into a scalar.

## 2365 C Syntax

```

2366      // scalar value
2367      GrB_Info GrB_Matrix_extractElement(<type>          *val,
2368                                         const GrB_Matrix A,
2369                                         GrB_Index      row_index,
2370                                         GrB_Index      col_index);
2371
2372      // GraphBLAS scalar
  
```

```

2373         GrB_Info GrB_Matrix_extractElement(GrB_Scalar      s,
2374                                           const GrB_Matrix A,
2375                                           GrB_Index      row_index,
2376                                           GrB_Index      col_index);
2377

```

## 2378 Parameters

2379     **val or s (INOUT)** An existing scalar whose domain is compatible with the domain of matrix  
2380     A. On successful return, this scalar holds the result of the extract. Any previous  
2381     value stored in **val** or **s** is overwritten.

2382     **A (IN)** The GraphBLAS matrix from which an element is extracted.

2383     **row\_index (IN)** The row index of location in **A** to extract.

2384     **col\_index (IN)** The column index of location in **A** to extract.

## 2385 Return Values

2386     **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2387     cessfully. This indicates that the compatibility tests on dimensions  
2388     and domains for the input arguments passed successfully, and the  
2389     output scalar, **val** or **s**, has been computed and is ready to be used  
2390     in the next method of the sequence.

2391     **GrB\_NO\_VALUE** When using the transparent scalar, **val**, this is returned when there  
2392     is no stored value at specified location.

2393     **GrB\_PANIC** Unknown internal error.

2394     **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2395     GraphBLAS objects (input or output) is in an invalid state caused  
2396     by a previous execution error. Call **GrB\_error()** to access any error  
2397     messages generated by the implementation.

2398     **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2399     **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, or scalar, **s**, has not been initialized by  
2400     a call to a corresponding constructor.

2401     **GrB\_NULL\_POINTER** **val** pointer is **NULL**.

2402     **GrB\_INVALID\_INDEX** **row\_index** or **col\_index** is outside the allowable range (i.e. less than  
2403     zero or greater than or equal to **nrows(A)** or **ncols(A)**, respec-  
2404     tively).

2405     **GrB\_DOMAIN\_MISMATCH** The domains of the matrix and scalar are incompatible.

## 2406 Description

2407 First, the scalar and input matrix are tested for domain compatibility as follows:  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(\mathbf{s})$   
 2408 must be compatible with  $\mathbf{D}(\mathbf{A})$ . Two domains are compatible with each other if values from  
 2409 one domain can be cast to values in the other domain as per the rules of the C language. In  
 2410 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
 2411 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
 2412 `GrB_Matrix_extractElement` ends and the domain mismatch error listed above is returned.

2413 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2414 \quad & 0 \leq \text{row\_index} < \mathbf{nrows}(\mathbf{A}), \\ & 0 \leq \text{col\_index} < \mathbf{ncols}(\mathbf{A}) \end{aligned}$$

2415 If either condition is violated, execution of `GrB_Matrix_extractElement` ends and the invalid index  
 2416 error listed above is returned.

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

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

2417 If  $(\text{row\_index}, \text{col\_index}) \in \mathbf{ind}(\mathbf{A})$ , then the corresponding value from  $\mathbf{A}$  is copied into  $\mathbf{s}$  or  $\text{val}$   
 2418 with casting as necessary. If  $(\text{row\_index}, \text{col\_index}) \notin \mathbf{ind}(\mathbf{A})$ , then one of the follow occurs  
 2419 depending on output scalar type:

- 2420 • The GraphBLAS scalar,  $\mathbf{s}$ , is cleared and `GrB_SUCCESS` is returned.
- 2421 • The non-opaque scalar,  $\text{val}$ , is unchanged, and `GrB_NO_VALUE` is returned.

2422 When using the non-opaque scalar variant ( $\text{val}$ ) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`  
 2423 mode, the new contents of  $\text{val}$  are as defined above if the method exits with return value `GrB_SUCCESS`  
 2424 or `GrB_NO_VALUE`.

2425 When using the GraphBLAS scalar variant ( $\mathbf{s}$ ) with a `GrB_SUCCESS` return value, the method  
 2426 exits and the new contents of  $\mathbf{s}$  is as defined above and fully computed in `GrB_BLOCKING` mode.  
 2427 In `GrB_NONBLOCKING` mode, the new contents of  $\mathbf{s}$  is as defined above but may not be fully  
 2428 computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 2429 4.2.5.13 Matrix\_extractTuples: Extract tuples from a matrix

2430 Extract the contents of a GraphBLAS matrix into non-opaque data structures.

## 2431 C Syntax

```
2432      GrB_Info GrB_Matrix_extractTuples(GrB_Index      *row_indices,
2433                                     GrB_Index      *col_indices,
```

```

2434                                     <type>          *values,
2435                                     GrB_Index        *n,
2436                                     const GrB_Matrix  A);

```

## 2437 Parameters

2438     **row\_indices** (OUT) Pointer to an array of row indices that is large enough to hold all of the  
2439     row indices.

2440     **col\_indices** (OUT) Pointer to an array of column indices that is large enough to hold all of the  
2441     column indices.

2442     **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of  
2443     the stored values whose type is compatible with  $\mathbf{D}(\mathbf{A})$ .

2444     **n** (INOUT) Pointer to a value indicating (in input) the number of elements the **values**,  
2445     **row\_indices**, and **col\_indices** arrays can hold. Upon return, it will contain the  
2446     number of values written to the arrays.

2447     **A** (IN) An existing GraphBLAS matrix.

## 2448 Return Values

2449     **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2450     cessfully. This indicates that the compatibility tests on the input  
2451     argument passed successfully, and the output arrays, **indices** and  
2452     **values**, have been computed.

2453     **GrB\_PANIC** Unknown internal error.

2454     **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2455     GraphBLAS objects (input or output) is in an invalid state caused  
2456     by a previous execution error. Call **GrB\_error()** to access any error  
2457     messages generated by the implementation.

2458     **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2459     **GrB\_INSUFFICIENT\_SPACE** Not enough space in **row\_indices**, **col\_indices**, and **values** (as indi-  
2460     cated by the **n** parameter) to hold all of the tuples that will be  
2461     extracted.

2462     **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to  
2463     any matrix constructor.

2464     **GrB\_NULL\_POINTER** **row\_indices**, **col\_indices**, **values** or **n** pointer is NULL.

2465     **GrB\_DOMAIN\_MISMATCH** The domains of the **A** matrix and **values** array are incompatible  
2466     with one another.

## 2467 Description

2468 This method will extract all the tuples from the GraphBLAS matrix **A**. The values associated with  
2469 those tuples are placed in the **values** array, the column indices are placed in the **col\_indices** array,  
2470 and the row indices are placed in the **row\_indices** array. These output arrays are pre-allocated by  
2471 the user before calling this function such that each output array has enough space to hold at least  
2472 **GrB\_Matrix\_nvals(A)** elements.

2473 Upon return of this function, a pair of  $\{\text{row\_indices}[k], \text{col\_indices}[k]\}$  are unique for every valid  
2474  $k$ , but they are not required to be sorted in any particular order. Each tuple  $(i, j, A_{ij})$  in **A** is  
2475 unzipped and copied into a distinct  $k$ th location in output vectors:

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

2476 where  $0 \leq k < \text{GrB\_Matrix\_nvals}(v)$ . No gaps in output vectors are allowed; that is, if **row\_indices**[ $k$ ],  
2477 **col\_indices**[ $k$ ] and **values**[ $k$ ] exist upon return, so does **row\_indices**[ $j$ ], **col\_indices**[ $j$ ] and **values**[ $j$ ] for  
2478 all  $j$  such that  $0 \leq j < k$ .

2479 Note that if the value in **n** on input is less than the number of values contained in the matrix **A**,  
2480 then a **GrB\_INSUFFICIENT\_SPACE** error is returned since it is undefined which subset of values  
2481 would be extracted.

2482 In both **GrB\_BLOCKING** mode **GrB\_NONBLOCKING** mode if the method exits with return value  
2483 **GrB\_SUCCESS**, the new contents of the arrays **row\_indices**, **col\_indices** and **values** are as defined  
2484 above.

2485 **4.2.5.14 Matrix\_exportHint: Provide a hint as to which storage format might be most**  
2486 **efficient for exporting a matrix**

## 2487 C Syntax

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

## 2488 Parameters

2489 **hint** (OUT) Pointer to a value of type **GrB\_Format**.

2490 **A** (IN) A GraphBLAS matrix object.

## 2491 Return Values

2492 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2493 cessfully and the value of **hint** has been set.

2494 **GrB\_PANIC** Unknown internal error.

2495           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
 2496                                   opaque GraphBLAS objects (input or output) is in an invalid  
 2497                                   state caused by a previous execution error. Call GrB\_error() to  
 2498                                   access any error messages generated by the implementation.

2499           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2500           GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
 2501                                   any matrix constructor.

2502           GrB\_NULL\_POINTER hint is NULL.

2503           GrB\_NO\_VALUE If the implementation does not have a preferred format, it may  
 2504                                   return the value GrB\_NO\_VALUE.

## 2505 Description

2506 Given a GraphBLAS matrix A, provide a hint as to which format might be most efficient for  
 2507 exporting the matrix A. GraphBLAS implementations might return the current storage format of  
 2508 the matrix, or the format to which it could most efficiently be exported. However, implementations  
 2509 are free to return any value for format defined in Section 3.5.3.1. Note that an implementation is  
 2510 free to refuse to provide a format hint, returning GrB\_NO\_VALUE.

## 2511 4.2.5.15 Matrix\_exportSize: Return the array sizes necessary to export a GraphBLAS 2512 matrix object

## 2513 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);

```

## 2514 Parameters

2515           n\_indptr (OUT) Pointer to a value of type GrB\_Index.

2516           n\_indices (OUT) Pointer to a value of type GrB\_Index.

2517           n\_values (OUT) Pointer to a value of type GrB\_Index.

2518           format (IN) a value indicating the format in which the matrix will be exported, as defined  
 2519                                   in Section 3.5.3.1.

2520           A (IN) A GraphBLAS matrix object.



## 2521 Return Values

2522           GrB\_SUCCESS In blocking mode or non-blocking mode, the operation com-  
2523                       pleted successfully. This indicates that the API checks for the  
2524                       input arguments passed successfully, and the number of elements  
2525                       necessary for the export buffers have been written to `n_indptr`,  
2526                       `n_indices`, and `n_values`, respectively.

2527           GrB\_PANIC Unknown internal error.

2528           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
2529                               opaque GraphBLAS objects (input or output) is in an invalid  
2530                               state caused by a previous execution error. Call `GrB_error()` to  
2531                               access any error messages generated by the implementation.

2532           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2533           GrB\_UNINITIALIZED\_OBJECT The GraphBLAS Matrix, `A`, has not been initialized by a call to  
2534                                       any matrix constructor.

2535           GrB\_NULL\_POINTER `n_indptr`, `n_indices`, or `n_values` is NULL.

## 2536 Description

2537 Given a matrix `A`, returns the required capacities of arrays `values`, `indptr`, and `indices` necessary to  
2538 export the matrix in the format specified by `format`. The output values `n_values`, `n_indptr`, and  
2539 `indices` will contain the corresponding sizes of the arrays (in number of elements) that must be  
2540 allocated to hold the exported matrix. The argument `format` can be chosen arbitrarily by the user  
2541 as one of the values defined in Section 3.5.3.1.

### 2542 4.2.5.16 Matrix\_export: Export a GraphBLAS matrix to a pre-defined format

## 2543 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);
```

## 2544 Parameters

- 2545        **indptr** (INOUT) Pointer to an array that will hold row or column offsets, or row in-  
2546        dices, depending on the value of **format**. It must be large enough to hold at  
2547        least **n\_indptr** elements of type **GrB\_Index**, where **n\_indices** was returned from  
2548        **GrB\_Matrix\_exportSize()** method.
- 2549        **indices** (INOUT) Pointer to an array that will hold row or column indices of the elements  
2550        in **values**, depending on the value of **format**. It must be large enough to hold at  
2551        least **n\_indices** elements of type **GrB\_Index**, where **n\_indices** was returned from  
2552        **GrB\_Matrix\_exportSize()** method.
- 2553        **values** (INOUT) Pointer to an array that will hold stored values. The type of ele-  
2554        ment must match the type of the values stored in **A**. It must be large enough  
2555        to hold at least **n\_values** elements of that type, where **n\_values** was returned from  
2556        **GrB\_Matrix\_exportSize**.
- 2557        **n\_indptr** (INOUT) Pointer to a value indicating (on input) the number of elements the **indptr**  
2558        array can hold. Upon return, it will contain the number of elements written to the  
2559        array.
- 2560        **n\_indices** (INOUT) Pointer to a value indicating (on input) the number of elements the **indices**  
2561        array can hold. Upon return, it will contain the number of elements written to the  
2562        array.
- 2563        **n\_values** (INOUT) Pointer to a value indicating (on input) the number of elements the **values**  
2564        array can hold. Upon return, it will contain the number of elements written to the  
2565        array.
- 2566        **format** (IN) a value indicating the format in which the matrix will be exported, as defined  
2567        in Section 3.5.3.1.
- 2568        **A** (IN) A GraphBLAS matrix object.

## 2569 Return Values

- 2570        **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2571        cessfully. This indicates that the compatibility tests on the input  
2572        argument passed successfully, and the output arrays, **indptr**, **in-**  
2573        **dices** and **values**, have been computed.
- 2574        **GrB\_PANIC** Unknown internal error.
- 2575        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
2576        opaque GraphBLAS objects (input or output) is in an invalid  
2577        state caused by a previous execution error. Call **GrB\_error()** to  
2578        access any error messages generated by the implementation.
- 2579        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2580       GrB\_INSUFFICIENT\_SPACE Not enough space in `indptr`, `indices`, and/or `values` (as indicated  
2581                                   by the corresponding `n_*` parameter) to hold all of the corre-  
2582                                   sponding elements that will be extacted.

2583       GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to  
2584                                   any matrix constructor.

2585       GrB\_NULL\_POINTER `indptr`, `indices`, `values` `n_indptr`, `n_indices`, `n_values` pointer is  
2586                                   NULL.

2587       GrB\_DOMAIN\_MISMATCH The domain of `A` does not match with the type of `values`.

## 2588 Description

2589 Given a matrix `A`, this method exports the contents of the matrix into one of the pre-defined  
2590 `GrB_Format` formats from Section 3.5.3.1. The user-allocated arrays pointed to by `indptr`, `indices`,  
2591 and `values` must be at least large enough to hold the corresponding number of elements returned  
2592 by calling `GrB_Matrix_exportSize`. The value of `format` can be chosen arbitrarily, but a call to  
2593 `GrB_Matrix_exportHint` may suggest a format that results in the most efficient export. Details  
2594 of the contents of `indptr`, `indices`, and `values` corresponding to each supported format is given in  
2595 Appendix B.

### 2596 4.2.5.17 Matrix\_import: Import a matrix into a GraphBLAS object

## 2597 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);

```

## 2598 Parameters

2599       `A` (INOUT) On a successful return, contains a handle to the newly created Graph-  
2600                   BLAS matrix.

2601       `d` (IN) The type corresponding to the domain of the matrix being created. Can be  
2602           one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
2603           GraphBLAS type.

2604        **nrows** (IN) Integer value holding the number of rows in the matrix.

2605        **ncols** (IN) Integer value holding the number of columns in the matrix.

2606        **indptr** (IN) Pointer to an array of row or column offsets, or row indices, depending on the  
2607                value of **format**.

2608        **indices** (IN) Pointer to an array row or column indices of the elements in **values**, depending  
2609                on the value of **format**.

2610        **values** (IN) Pointer to an array of values. Type must match the type of **d**.

2611        **n\_indptr** (IN) Integer value holding the number of elements in the array pointed to by **indptr**.

2612        **n\_indices** (IN) Integer value holding the number of elements in the array pointed to by **indices**.

2613        **n\_values** (IN) Integer value holding the number of elements in the array pointed to by **values**.

2614        **format** (IN) a value indicating the format of the matrix being imported, as defined in  
2615                Section 3.5.3.1.

## 2616    **Return Values**

2617        **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2618                blocking mode, this indicates that the API checks for the input  
2619                arguments passed successfully and the input arrays have been  
2620                consumed. Either way, output matrix **A** is ready to be used in  
2621                the next method of the sequence.

2622        **GrB\_PANIC** Unknown internal error.

2623        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2624        **GrB\_UNINITIALIZED\_OBJECT** The **GrB\_Type** object has not been initialized by a call to **GrB\_Type\_new**  
2625                (needed for user-defined types).

2626        **GrB\_NULL\_POINTER** **A**, **indptr**, **indices** or **values** pointer is **NULL**.

2627        **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **indptr** or **indices** is outside the allowed range for indices  
2628                in **A** and or the size of **values**, **n\_values**, depending on the value  
2629                of **format**.

2630        **GrB\_INVALID\_VALUE** **nrows** or **ncols** is zero or outside the range of the type **GrB\_Index**.

2631        **GrB\_DOMAIN\_MISMATCH** The domain given in parameter **d** does not match the element  
2632                type of **values**.

## 2633 Description

2634 Creates a new matrix **A** of domain **D**(d) and dimension **nrows**  $\times$  **ncols**. The new GraphBLAS  
2635 matrix will be filled with the contents of the matrix pointed to by **indptr**, and **indices**, and **values**.  
2636 The method returns a handle to the new matrix in **A**. The structure of the data being imported is  
2637 defined by **format**, which must be equal to one of the values defined in Section 3.5.3.1. Details of  
2638 the contents of **indptr**, **indices** and **values** for each supported format is given in Appendix B.

2639 It is not an error to call this method more than once on the same output matrix; however, the  
2640 handle to the previously created object will be overwritten.

## 2641 4.2.5.18 Matrix\_serializeSize: Compute the serialize buffer size

2642 Compute the buffer size (in bytes) necessary to serialize a GrB\_Matrix using GrB\_Matrix\_serialize.

## 2643 C Syntax

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

## 2644 Parameters

2645 **size** (OUT) Pointer to GrB\_Index value where size in bytes of serialized object will be  
2646 written.

2647 **A** (IN) A GraphBLAS matrix object.

## 2648 Return Values

2649 **GrB\_SUCCESS** The operation completed successfully and the value pointed to  
2650 by **\*size** has been computed and is ready to use.

2651 **GrB\_PANIC** Unknown internal error.

2652 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2653 **GrB\_NULL\_POINTER** **size** is NULL.

## 2654 Description

2655 Returns the size in bytes of the data buffer necessary to serialize the GraphBLAS matrix object **A**.  
2656 Users may then allocate a buffer of **size** bytes to pass as a parameter to GrB\_Matrix\_serialize.

2657 **4.2.5.19 Matrix\_serialize: Serialize a GraphBLAS matrix.**

2658 Serialize a GraphBLAS Matrix object into an opaque stream of bytes.

2659 **C Syntax**

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

2660 **Parameters**

2661 **serialized\_data** (INOUT) Pointer to the preallocated buffer where the serialized matrix will be  
2662 written.

2663 **serialized\_size** (INOUT) On input, the size in bytes of the buffer pointed to by **serialized\_data**.  
2664 On output, the number of bytes written to **serialized\_data**.

2665 **A** (IN) A GraphBLAS matrix object.

2666 **Return Values**

2667 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2668 cessfully. This indicates that the compatibility tests on the in-  
2669 put argument passed successfully, and the output buffer **serial-  
2670 ized\_data** and **serialized\_size**, have been computed and are ready  
2671 to use.

2672 **GrB\_PANIC** Unknown internal error.

2673 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
2674 opaque GraphBLAS objects (input or output) is in an invalid  
2675 state caused by a previous execution error. Call **GrB\_error()** to  
2676 access any error messages generated by the implementation.

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

2678 **GrB\_NULL\_POINTER** **serialized\_data** or **serialize\_size** is NULL.

2679 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to  
2680 any matrix constructor.

2681 **GrB\_INSUFFICIENT\_SPACE** The size of the buffer **serialized\_data** (provided as an input **seri-  
2682 alized\_size**) was not large enough.

## 2683 Description

2684 Serializes a GraphBLAS matrix object to an opaque buffer. To guarantee successful execution,  
2685 the size of the buffer pointed to by `serialized_data`, provided as an input by `serialized_size`, must  
2686 be of at least the number of bytes returned from `GrB_Matrix_serializeSize`. The actual size of the  
2687 serialized matrix written to `serialized_data` is provided upon completion as an output written to  
2688 `serialized_size`.

2689 The contents of the serialized buffer are implementation defined. Thus, a serialized matrix created  
2690 with one library implementation is not necessarily valid for deserialization with another implemen-  
2691 tation.

### 2692 4.2.5.20 Matrix\_deserialize: Deserialize a GraphBLAS matrix.

2693 Construct a new GraphBLAS matrix from a serialized object.

## 2694 C Syntax

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

## 2695 Parameters

2696 A (INOUT) On a successful return, contains a handle to the newly created Graph-  
2697 BLAS matrix.

2698 d (IN) the type of the matrix that was serialized in `serialized_data`.

2699 `serialized_data` (IN) a pointer to a serialized GraphBLAS matrix created with `GrB_Matrix_serialize`.

2700 `serialized_size` (IN) the size of the buffer pointed to by `serialized_data` in bytes.

## 2701 Return Values

2702 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2703 blocking mode, this indicates that the API checks for the input  
2704 arguments passed successfully. Either way, output matrix A is  
2705 ready to be used in the next method of the sequence.

2706 GrB\_PANIC Unknown internal error.

2707 GrB\_INVALID\_OBJECT This is returned if `serialized_data` is invalid or corrupted.

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

2709 GrB\_UNINITIALIZED\_OBJECT The GrB\_Type object has not been initialized by a call to GrB\_Type\_new  
2710 (needed for user-defined types).

2711 GrB\_NULL\_POINTER serialized\_data or A is NULL.

2712 GrB\_DOMAIN\_MISMATCH The type given in d does not match the type of the matrix  
2713 serialized in serialized\_data.

## 2714 Description

2715 Creates a new matrix **A** using the serialized matrix object pointed to by `serialized_data`. The object  
2716 pointed to by `serialized_data` must have been created using the method `GrB_Matrix_serialize`. The  
2717 domain of the matrix is given as an input in `d`, which must match the domain of the matrix serialized  
2718 in `serialized_data`. Note that for user-defined types, only the size of the type will be checked.

2719 Since the format of a serialized matrix is implementation-defined, it is not guaranteed that a matrix  
2720 serialized in one library implementation can be deserialized by another.

2721 It is not an error to call this method more than once on the same output matrix; however, the  
2722 handle to the previously created object will be overwritten.

## 2723 4.2.6 Descriptor methods

2724 The methods in this section create and set values in descriptors. A descriptor is an opaque Graph-  
2725 BLAS object the values of which are used to modify the behavior of GraphBLAS operations.

### 2726 4.2.6.1 Descriptor\_new: Create new descriptor

2727 Creates a new (empty or default) descriptor.

## 2728 C Syntax

2729 GrB\_Info GrB\_Descriptor\_new(GrB\_Descriptor \*desc);

## 2730 Parameters

2731 desc (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2732 descriptor.

## 2733 Return Value

2734 GrB\_SUCCESS The method completed successfully.

2735 GrB\_PANIC unknown internal error.



2736        GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

2737        GrB\_NULL\_POINTER desc pointer is NULL.

## 2738    **Description**

2739    Creates a new descriptor object and returns a handle to it in desc. A newly created descriptor can  
2740    be populated by calls to Descriptor\_set.

2741    It is not an error to call this method more than once on the same variable; however, the handle to  
2742    the previously created object will be overwritten.

### 2743    **4.2.6.2    Descriptor\_set: Set content of descriptor**

2744    Sets the content for a field for an existing descriptor.

## 2745    **C Syntax**

```
2746        GrB_Info GrB_Descriptor_set(GrB_Descriptor        desc,  
2747                                    GrB_Desc_Field        field,  
2748                                    GrB_Desc_Value        val);
```

## 2749    **Parameters**

2750        desc (IN) An existing GraphBLAS descriptor to be modified.

2751        field (IN) The field being set.

2752        val (IN) New value for the field being set.

## 2753    **Return Values**

2754        GrB\_SUCCESS operation completed successfully.

2755        GrB\_PANIC unknown internal error.

2756        GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

2757        GrB\_UNINITIALIZED\_OBJECT the desc parameter has not been initialized by a call to new.

2758        GrB\_INVALID\_VALUE invalid value set on the field, or invalid field.

## 2759 Description

2760 For a given descriptor, the `GrB_Descriptor_set` method can be called for each field in the descriptor  
2761 to set the value associated with that field. Valid values for the `field` parameter include the following:

2762 `GrB_OUTP` refers to the output parameter (result) of the operation.

2763 `GrB_MASK` refers to the mask parameter of the operation.

2764 `GrB_INP0` refers to the first input parameters of the operation (matrices and vectors).

2765 `GrB_INP1` refers to the second input parameters of the operation (matrices and vectors).

2766 Valid values for the `val` parameter are:

2767 `GrB_STRUCTURE` Use only the structure of the stored values of the corresponding mask  
2768 (`GrB_MASK`) parameter.

2769 `GrB_COMP` Use the complement of the corresponding mask (`GrB_MASK`) param-  
2770 eter. When combined with `GrB_STRUCTURE`, the complement of the  
2771 structure of the mask is used without evaluating the values stored.

2772 `GrB_TRAN` Use the transpose of the corresponding matrix parameter (valid for input  
2773 matrix parameters only).

2774 `GrB_REPLACE` When assigning the masked values to the output matrix or vector, clear  
2775 the matrix first (or clear the non-masked entries). The default behavior  
2776 is to leave non-masked locations unchanged. Valid for the `GrB_OUTP`  
2777 parameter only.

2778 Descriptor values can only be set, and once set, cannot be cleared. As, in the case of `GrB_MASK`,  
2779 multiple values can be set and all will apply (for example, both `GrB_COMP` and `GrB_STRUCTURE`).  
2780 A value for a given field may be set multiple times but will have no additional effect. Fields that  
2781 have no values set result in their default behavior, as defined in Section 3.6.

## 2782 4.2.7 free: Destroy an object and release its resources

2783 Destroys a previously created GraphBLAS object and releases any resources associated with the  
2784 object.

## 2785 C Syntax

2786 `GrB_Info GrB_free(<GrB_Object> *obj);`

## 2787 Parameters

2788       obj (INOUT) An existing GraphBLAS object to be destroyed. The object must have  
2789       been created by an explicit call to a GraphBLAS constructor. It can be any of the  
2790       opaque GraphBLAS objects such as matrix, vector, descriptor, semiring, monoid,  
2791       binary op, unary op, or type. On successful completion of GrB\_free, obj behaves  
2792       as an uninitialized object.

## 2793 Return Values

2794       GrB\_SUCCESS operation completed successfully

2795       GrB\_PANIC unknown internal error. If this return value is encountered when  
2796       in nonblocking mode, the error responsible for the panic condition  
2797       could be from any method involved in the computation of the input  
2798       object. The GrB\_error() method should be called for additional  
2799       information.

## 2800 Description

2801 GraphBLAS objects consume memory and other resources managed by the GraphBLAS runtime  
2802 system. A call to GrB\_free frees those resources so they are available for use by other GraphBLAS  
2803 objects.

2804 The parameter passed into GrB\_free is a handle referencing a GraphBLAS opaque object of a data  
2805 type from table 2.1. The object must have been created by an explicit call to a GraphBLAS con-  
2806 structor. The behavior of a program that calls GrB\_free on a pre-defined object is implementation  
2807 defined.

2808 After the GrB\_free method returns, the object referenced by the input handle is destroyed and the  
2809 handle has the value GrB\_INVALID\_HANDLE. The handle can be used in subsequent GraphBLAS  
2810 methods but only after the handle has been reinitialized with a call the the appropriate \_new or  
2811 \_dup method.

2812 Note that unlike other GraphBLAS methods, calling GrB\_free with an object with an invalid handle  
2813 is legal. The system may attempt to free resources that might be associated with that object, if  
2814 possible, and return normally.

2815 When using GrB\_free it is possible to create a dangling reference to an object. This would occur  
2816 when a handle is assigned to a second variable of the same opaque type. This creates two handles  
2817 that reference the same object. If GrB\_free is called with one of the variables, the object is destroyed  
2818 and the handle associated with the other variable no longer references a valid object. This is not an  
2819 error condition that the implementation of the GraphBLAS API can be expected to catch, hence  
2820 programmers must take care to prevent this situation from occurring.

#### 2821 4.2.8 wait: Return once an object is either *complete* or *materialized*

2822 Wait until method calls in a sequence put an object into a state of *completion* or *materialization*.

#### 2823 C Syntax

```
2824 GrB_Info GrB_wait(GrB_Object obj, GrB_WaitMode mode);
```

#### 2825 Parameters

2826 **obj** (INOUT) An existing GraphBLAS object. The object must have been created by an  
2827 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS  
2828 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,  
2829 or type. On successful return of `GrB_wait`, the **obj** can be safely read from another  
2830 thread (completion) or all computing to produce **obj** by all GraphBLAS operations  
2831 in its sequence have finished (materialization).

2832 **mode** (IN) Set's the mode for `GrB_wait` for whether it is waiting for **obj** to be in the  
2833 state of *completion* or *materialization*. Acceptable values are `GrB_COMPLETE` or  
2834 `GrB_MATERIALIZE`.

#### 2835 Return values

2836 `GrB_SUCCESS` operation completed successfully.

2837 `GrB_INDEX_OUT_OF_BOUNDS` an index out-of-bounds execution error happened during com-  
2838 pletion of pending operations.

2839 `GrB_OUT_OF_MEMORY` and out-of-memory execution error happened during completion  
2840 of pending operations.

2841 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`,  
2842 or other constructor, method.

2843 `GrB_PANIC` unknown internal error.

2844 `GrB_INVALID_VALUE` method called with a `GrB_WaitMode` other than `GrB_COMPLETE`  
2845 `GrB_MATERIALIZE`.

#### 2846 Description

2847 On successful return from `GrB_wait()`, the input object, **obj** is in one of two states depending on  
2848 the mode of `GrB_wait`:

- 2849 • *complete*: `obj` can be used in a happens-before relation, so in a properly synchronized program  
2850 it can be safely used as an IN or INOUT parameter in a GraphBLAS method call from another  
2851 thread. This result occurs when the mode parameter is set to `GrB_COMPLETE`.
- 2852 • *materialized*: `obj` is *complete*, but in addition, no further computing will be carried out on  
2853 behalf of `obj` and error information is available. This result occurs when the mode parameter  
2854 is set to `GrB_MATERIALIZE`.

2855 Since in blocking mode OUT or INOUT parameters to any method call are materialized upon return,  
2856 `GrB_wait(obj,mode)` has no effect when called in blocking mode.

2857 In non-blocking mode, the status of any pending method calls, other than those associated with pro-  
2858 ducing the *complete* or *materialized* state of `obj`, are not impacted by the call to `GrB_wait(obj,mode)`.  
2859 Methods in the sequence for `obj`, however, most likely would be impacted by a call to `GrB_wait(obj,mode)`;  
2860 especially in the case of the *materialized* mode for which any computing on behalf of `obj` must be  
2861 finished prior to the return from `GrB_wait(obj,mode)`.

#### 2862 4.2.9 error: Retrieve an error string

2863 Retrieve an error-message about any errors encountered during the processing associated with an  
2864 object.

### 2865 C Syntax

```
2866      GrB_Info GrB_error(const char      **error,
2867                        const GrB_Object  obj);
```

#### 2868 Parameters

2869 `error` (OUT) A pointer to a null-terminated string. The contents of the string are im-  
2870 plementation defined.

2871 `obj` (IN) An existing GraphBLAS object. The object must have been created by an  
2872 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS  
2873 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,  
2874 or type.

#### 2875 Return value

2876 `GrB_SUCCESS` operation completed successfully.

2877 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`,  
2878 or other constructor, method.

2879 `GrB_PANIC` unknown internal error.

## Description

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

## 4.3 GraphBLAS operations

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

### Domains and Casting

A GraphBLAS operation is only valid when the domains of the GraphBLAS objects are mathematically consistent. The C programming language defines implicit casts between built-in data types. For example, floats, doubles, and ints can be freely mixed according to the rules defined for implicit casts. It is the responsibility of the user to assure that these casts are appropriate for the algorithm in question. For example, a cast to int implies truncation of a floating point type. Depending on the operation, this truncation error could lead to erroneous results. Furthermore, casting a wider type onto a narrower type can lead to overflow errors. The GraphBLAS operations do not attempt to protect a user from these sorts of errors.

When user-defined types are involved, however, GraphBLAS requires strict equivalence between types and no casting is supported. If GraphBLAS detects these mismatches, it will return a domain mismatch error.

### Dimensions and Transposes

GraphBLAS operations also make assumptions about the numbers of dimensions and the sizes of vectors and matrices in an operation. An operation will test these sizes and report an error if they are not *shape compatible*. For example, when multiplying two matrices,  $\mathbf{C} = \mathbf{A} \times \mathbf{B}$ , the number of rows of  $\mathbf{C}$  must equal the number of rows of  $\mathbf{A}$ , the number of columns of  $\mathbf{A}$  must match the number of rows of  $\mathbf{B}$ , and the number of columns of  $\mathbf{C}$  must match the number of columns of  $\mathbf{B}$ . This is the behavior expected given the mathematical definition of the operations.

For most of the GraphBLAS operations involving matrices, an optional descriptor can modify the matrix associated with an input GraphBLAS matrix object. For example, if an input matrix is an

Table 4.1: A mathematical notation for the fundamental GraphBLAS operations supported in this specification. Input matrices  $\mathbf{A}$  and  $\mathbf{B}$  may be optionally transposed (not shown). Use of an optional accumulate with existing values in the output object is indicated with  $\odot$ . Use of optional write masks and replace flags are indicated as  $\mathbf{C}\langle\mathbf{M}, r\rangle$  when applied to the output matrix,  $\mathbf{C}$ . The mask controls which values resulting from the operation on the right-hand side are written into the output object (complement and structure flags are not shown). The “replace” option, indicated by specifying the  $r$  flag, means that all values in the output object are removed prior to assignment. If “replace” is not specified, only the values/locations computed on the right-hand side and allowed by the mask will be written to the output (“merge” mode).

Operation Name	Mathematical Notation		
mxm	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A} \oplus . \otimes \mathbf{B}$
mxv	$\mathbf{w}\langle\mathbf{m}, r\rangle$	=	$\mathbf{w} \odot \mathbf{A} \oplus . \otimes \mathbf{u}$
vxm	$\mathbf{w}^T\langle\mathbf{m}^T, r\rangle$	=	$\mathbf{w}^T \odot \mathbf{u}^T \oplus . \otimes \mathbf{A}$
eWiseMult	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A} \otimes \mathbf{B}$
	$\mathbf{w}\langle\mathbf{m}, r\rangle$	=	$\mathbf{w} \odot \mathbf{u} \otimes \mathbf{v}$
eWiseAdd	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A} \oplus \mathbf{B}$
	$\mathbf{w}\langle\mathbf{m}, r\rangle$	=	$\mathbf{w} \odot \mathbf{u} \oplus \mathbf{v}$
extract	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A}(i, j)$
	$\mathbf{w}\langle\mathbf{m}, r\rangle$	=	$\mathbf{w} \odot \mathbf{u}(i)$
assign	$\mathbf{C}\langle\mathbf{M}, r\rangle(i, j)$	=	$\mathbf{C}(i, j) \odot \mathbf{A}$
	$\mathbf{w}\langle\mathbf{m}, r\rangle(i)$	=	$\mathbf{w}(i) \odot \mathbf{u}$
reduce (row)	$\mathbf{w}\langle\mathbf{m}, r\rangle$	=	$\mathbf{w} \odot [\oplus_j \mathbf{A}(:, j)]$
reduce (scalar)	$s$	=	$s \odot [\oplus_{i,j} \mathbf{A}(i, j)]$
	$s$	=	$s \odot [\oplus_i \mathbf{u}(i)]$
apply	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot f_u(\mathbf{A})$
	$\mathbf{w}\langle\mathbf{m}, r\rangle$	=	$\mathbf{w} \odot f_u(\mathbf{u})$
apply(indexop)	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot f_i(\mathbf{A}, \text{ind}(\mathbf{A}), s)$
	$\mathbf{w}\langle\mathbf{m}, r\rangle$	=	$\mathbf{w} \odot f_i(\mathbf{u}, \text{ind}(\mathbf{u}), s)$
select	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A}\langle f_i(\mathbf{A}, \text{ind}(\mathbf{A}), s) \rangle$
	$\mathbf{w}\langle\mathbf{m}, r\rangle$	=	$\mathbf{w} \odot \mathbf{u}\langle f_i(\mathbf{u}, \text{ind}(\mathbf{u}), s) \rangle$
transpose	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A}^T$
kronecker	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A} \otimes \mathbf{B}$

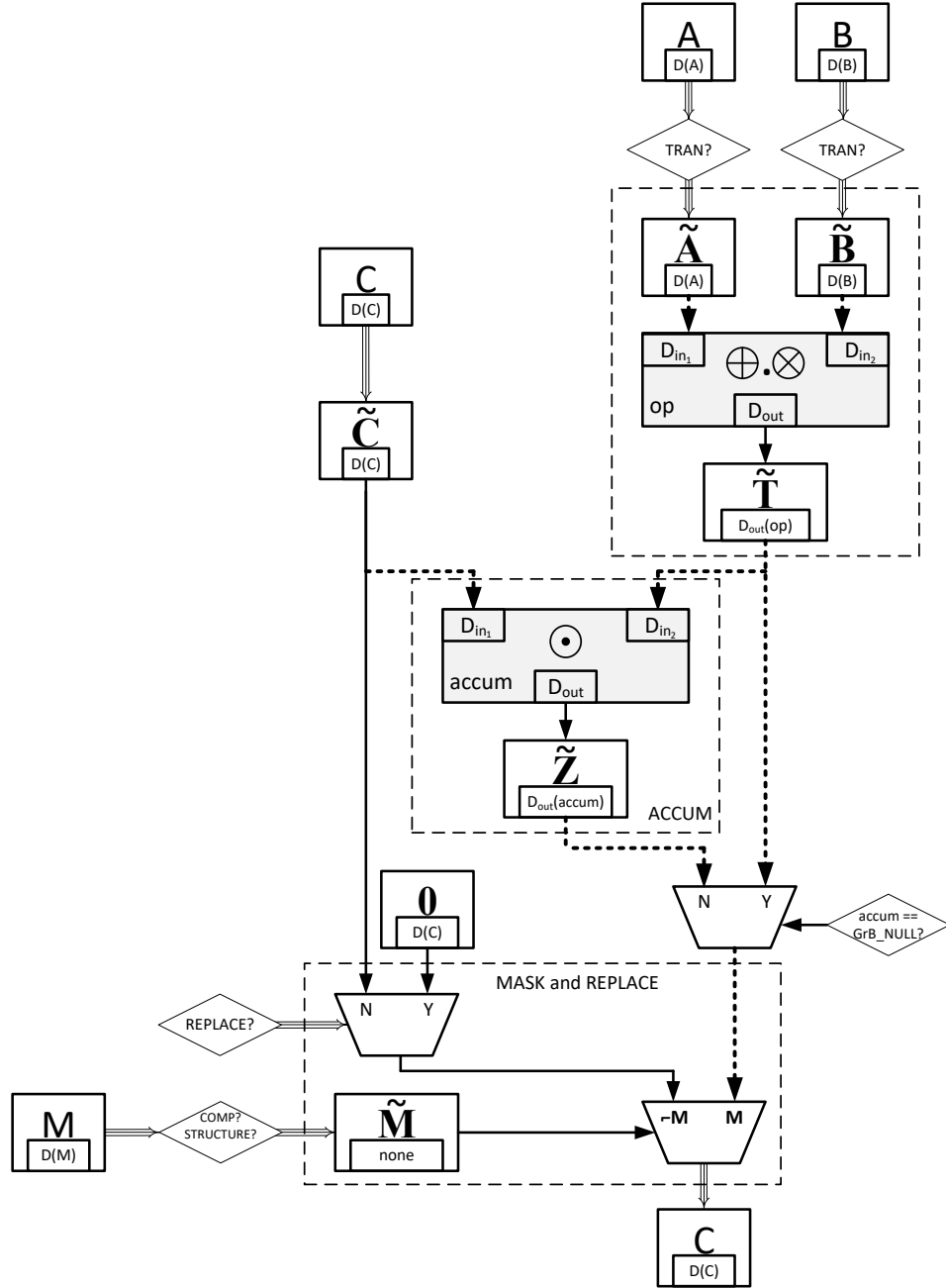


Figure 4.1: Flowchart for the GraphBLAS operations. Although shown specifically for the mxm operation, many elements are common to all operations: such as the “ACCUM” and “MASK and REPLACE” blocks. The triple arrows ( $\Rightarrow$ ) denote where “as if copy” takes place (including both collections and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



argument to a GraphBLAS operation and the associated descriptor indicates the transpose option, then the operation occurs as if on the transposed matrix. In this case, the relationships between the sizes in each dimension shift in the mathematically expected way.

## Masks: Structure-only, Complement, and Replace

When a GraphBLAS operation supports the use of an optional mask, that mask is specified through a GraphBLAS vector (for one-dimensional masks) or a GraphBLAS matrix (for two-dimensional masks). When a mask is used and the `GrB_STRUCTURE` descriptor value is not set, it is applied to the result from the operation wherever the stored values in the mask evaluate to true. If the `GrB_STRUCTURE` descriptor is set, the mask is applied to the result from the operation wherever the mask as a stored value (regardless of that value). Wherever the mask is applied, the result from the operation is either assigned to the provided output matrix/vector or, if a binary accumulation operation is provided, the result is accumulated into the corresponding elements of the provided output matrix/vector.

Given a GraphBLAS vector  $\mathbf{v} = \langle D, N, \{(i, v_i)\} \rangle$ , a one-dimensional mask is derived for use in the operation as follows:

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

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

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

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

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

When the descriptor of an operation with a mask has specified that the `GrB_REPLACE` value is to be applied to the output (`GrB_OUTP`), then anywhere the mask is not true, the corresponding location in the output is cleared.

## Invalid and uninitialized objects

Upon entering a GraphBLAS operation, the first step is a check that all objects are valid and initialized. (Optional parameters can be set to `GrB_NULL`, which always counts as a valid object.) An invalid object is one that could not be computed due to a previous execution error. An uninitialized object is one that has not yet been created by a corresponding `new` or `dup` method. Appropriate error codes are returned if an object is not initialized (`GrB_UNINITIALIZED_OBJECT`) or invalid (`GrB_INVALID_OBJECT`).

2949 To support the detection of as many cases of uninitialized objects as possible, it is strongly rec-  
 2950 ommended to initialize all GraphBLAS objects to the predefined value `GrB_INVALID_HANDLE` at  
 2951 the point of their declaration, as shown in the following examples:

```
2952         GrB_Type          type = GrB_INVALID_HANDLE;
2953         GrB_Semiring       semiring = GrB_INVALID_HANDLE;
2954         GrB_Matrix        matrix = GrB_INVALID_HANDLE;
```

## 2955 Compliance

2956 We follow a *prescriptive* approach to the definition of the semantics of GraphBLAS operations.  
 2957 That is, for each operation we give a recipe for producing its outcome. Any implementation that  
 2958 produces the same outcome, and follows the GraphBLAS execution model (Section 2.5) and error  
 2959 model (Section 2.6) is a conforming implementation.

### 2960 4.3.1 mxm: Matrix-matrix multiply

2961 Multiplies a matrix with another matrix on a semiring. The result is a matrix.

## 2962 C Syntax

```
2963         GrB_Info GrB_mxm(GrB_Matrix          C,
2964                          const GrB_Matrix    Mask,
2965                          const GrB_BinaryOp   accum,
2966                          const GrB_Semiring   op,
2967                          const GrB_Matrix     A,
2968                          const GrB_Matrix     B,
2969                          const GrB_Descriptor desc);
```

## 2970 Parameters

2971 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
 2972 that may be accumulated with the result of the matrix product. On output, the  
 2973 matrix holds the results of the operation.

2974 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
 2975 stored into the output matrix C. The mask dimensions must match those of the  
 2976 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
 2977 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types  
 2978 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
 2979 dimensions of C), `GrB_NULL` should be specified.

2980 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
 2981 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
 2982 specified.

2983 **op** (IN) The semiring used in the matrix-matrix multiply.

2984 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
 2985 multiplication.

2986 **B** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
 2987 multiplication.

2988 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
 2989 should be specified. Non-default field/value pairs are listed as follows:  
 2990

Param	Field	Value	Description
<b>C</b>	<b>GrB_OUTP</b>	<b>GrB_REPLACE</b>	Output matrix <b>C</b> is cleared (all elements removed) before the result is stored in it.
<b>Mask</b>	<b>GrB_MASK</b>	<b>GrB_STRUCTURE</b>	The write mask is constructed from the structure (pattern of stored values) of the input <b>Mask</b> matrix. The stored values are not examined.
<b>Mask</b>	<b>GrB_MASK</b>	<b>GrB_COMP</b>	Use the complement of <b>Mask</b> .
<b>A</b>	<b>GrB_INP0</b>	<b>GrB_TRAN</b>	Use transpose of <b>A</b> for the operation.
<b>B</b>	<b>GrB_INP1</b>	<b>GrB_TRAN</b>	Use transpose of <b>B</b> for the operation.

## 2992 Return Values

2993 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 2994 blocking mode, this indicates that the compatibility tests on di-  
 2995 mensions and domains for the input arguments passed successfully.  
 2996 Either way, output matrix **C** is ready to be used in the next method  
 2997 of the sequence.

2998 **GrB\_PANIC** Unknown internal error.

2999 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 3000 GraphBLAS objects (input or output) is in an invalid state caused  
 3001 by a previous execution error. Call **GrB\_error()** to access any error  
 3002 messages generated by the implementation.

3003 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

3004 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 3005 a call to **new** (or **Matrix\_dup** for matrix parameters).

3006 **GrB\_DIMENSION\_MISMATCH** Mask and/or matrix dimensions are incompatible.

3007     GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
3008                               corresponding domains of the semiring or accumulation operator,  
3009                               or the mask's domain is not compatible with **bool** (in the case where  
3010                               desc[GrB\_MASK].GrB\_STRUCTURE is not set).

## 3011     **Description**

3012     GrB\_mxm computes the matrix product  $C = A \oplus . \otimes B$  or, if an optional binary accumulation operator  
3013     ( $\odot$ ) is provided,  $C = C \odot (A \oplus . \otimes B)$  (where matrices A and B can be optionally transposed).  
3014     Logically, this operation occurs in three steps:

3015         **Setup** The internal matrices and mask used in the computation are formed and their domains  
3016                     and dimensions are tested for compatibility.

3017         **Compute** The indicated computations are carried out.

3018         **Output** The result is written into the output matrix, possibly under control of a mask.

3019     Up to four argument matrices are used in the GrB\_mxm operation:

- 3020         1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3021         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)
- 3022         3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3023         4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3024     The argument matrices, the semiring, and the accumulation operator (if provided) are tested for  
3025     domain compatibility as follows:

- 3026         1. If Mask is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{Mask})$   
3027                 must be from one of the pre-defined types of Table 3.2.
- 3028         2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the semiring.
- 3029         3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the semiring.
- 3030         4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the semiring.
- 3031         5. If accum is not GrB\_NULL, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
3032                 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
3033                 of the accumulation operator.

3034     Two domains are compatible with each other if values from one domain can be cast to values in  
3035     the other domain as per the rules of the C language. In particular, domains from Table 3.2 are  
3036     all compatible with each other. A domain from a user-defined type is only compatible with itself.

3037 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the domain mismatch  
 3038 error listed above is returned.

3039 From the argument matrices, the internal matrices and mask used in the computation are formed  
 3040 ( $\leftarrow$  denotes copy):

- 3041 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 3042 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
  - 3043 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 3044  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 3045 (b) If `Mask  $\neq$  GrB_NULL`,
    - 3046 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
 3047  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 3048 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
 3049  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 3050 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 3051 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 3052 4. Matrix  $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}$ .

3053 The internal matrices and masks are checked for dimension compatibility. The following conditions  
 3054 must hold:

- 3055 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 3056 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 3057 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
- 3058 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$ .
- 3059 5.  $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$ .

3060 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the dimension mismatch  
 3061 error listed above is returned.

3062 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3063 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3064 We are now ready to carry out the matrix multiplication and any additional associated operations.  
 3065 We describe this in terms of two intermediate matrices:

- 3066 •  $\tilde{\mathbf{T}}$ : The matrix holding the product of matrices  $\tilde{\mathbf{A}}$  and  $\tilde{\mathbf{B}}$ .
- 3067 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

3068 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{B}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{B}}(:, j)) \neq \emptyset\} \rangle$  is created. The value of each of its elements is computed by

$$3070 \quad T_{ij} = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{B}}(:, j))} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{B}}(k, j)),$$

3071 where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring  $\mathbf{op}$ , respectively.

3072 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 3073 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 3074 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$3075 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3076 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
3077 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 3078 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 3079 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3080 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3081 \end{aligned}$$

3082 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

3084 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
3085 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
3086 mask which acts as a “write mask”.

- 3087 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
3088 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$3089 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3090 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
3091 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
3092 mask are unchanged:

$$3093 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3094 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
3095 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
3096 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
3097 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
3098 sequence.

### 3099 4.3.2 vxm: Vector-matrix multiply

3100 Multiplies a (row) vector with a matrix on an semiring. The result is a vector.

#### 3101 C Syntax

```
3102         GrB_Info GrB_vxm(GrB_Vector          w,  
3103                           const GrB_Vector    mask,  
3104                           const GrB_BinaryOp    accum,  
3105                           const GrB_Semiring    op,  
3106                           const GrB_Vector    u,  
3107                           const GrB_Matrix     A,  
3108                           const GrB_Descriptor  desc);
```

#### 3109 Parameters

3110 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3111 that may be accumulated with the result of the vector-matrix product. On output,  
3112 this vector holds the results of the operation.

3113 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3114 stored into the output vector **w**. The mask dimensions must match those of the  
3115 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3116 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3117 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
3118 dimensions of **w**), **GrB\_NULL** should be specified.

3119 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
3120 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
3121 specified.

3122 **op** (IN) Semiring used in the vector-matrix multiply.

3123 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the  
3124 multiplication.

3125 **A** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
3126 multiplication.

3127 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
3128 should be specified. Non-default field/value pairs are listed as follows:

3129

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.
A	GrB_INP1	GrB_TRAN	Use transpose of A for the operation.

3130

### 3131 Return Values

3132           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
3133                           blocking mode, this indicates that the compatibility tests on di-  
3134                           mensions and domains for the input arguments passed successfully.  
3135                           Either way, output vector w is ready to be used in the next method  
3136                           of the sequence.

3137           GrB\_PANIC Unknown internal error.

3138           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
3139                           GraphBLAS objects (input or output) is in an invalid state caused  
3140                           by a previous execution error. Call GrB\_error() to access any error  
3141                           messages generated by the implementation.

3142           GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

3143 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
3144                           a call to new (or dup for matrix or vector parameters).

3145 GrB\_DIMENSION\_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

3146           GrB\_DOMAIN\_MISMATCH The domains of the various vectors/matrices are incompatible with  
3147                           the corresponding domains of the semiring or accumulation opera-  
3148                           tor, or the mask's domain is not compatible with bool (in the case  
3149                           where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

### 3150 Description

3151 GrB\_vxm computes the vector-matrix product  $w^T = u^T \oplus . \otimes A$ , or, if an optional binary accu-  
3152                           mulation operator ( $\odot$ ) is provided,  $w^T = w^T \odot (u^T \oplus . \otimes A)$  (where matrix A can be optionally  
3153                           transposed). Logically, this operation occurs in three steps:

3154           **Setup** The internal vectors, matrices and mask used in the computation are formed and their  
3155                           domains/dimensions are tested for compatibility.

3156           **Compute** The indicated computations are carried out.



3157     **Output** The result is written into the output vector, possibly under control of a mask.

3158 Up to four argument vectors or matrices are used in the `GrB_vxm` operation:

- 3159     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3160     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3161     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3162     4.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

3163 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are  
3164 tested for domain compatibility as follows:

- 3165     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
3166         must be from one of the pre-defined types of Table 3.2.
- 3167     2.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the semiring.
- 3168     3.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$  of the semiring.
- 3169     4.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the semiring.
- 3170     5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
3171         of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$   
3172         of the accumulation operator.

3173 Two domains are compatible with each other if values from one domain can be cast to values in  
3174 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are  
3175 all compatible with each other. A domain from a user-defined type is only compatible with itself.  
3176 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the domain mismatch  
3177 error listed above is returned.

3178 From the argument vectors and matrices, the internal matrices and mask used in the computation  
3179 are formed ( $\leftarrow$  denotes copy):

- 3180     1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3181     2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 3182         (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 3183         (b) If `mask`  $\neq$  `GrB_NULL`,
    - 3184             i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 3185             ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}(\mathbf{mask})(i) = \mathbf{true})\} \rangle$ .
  - 3186         (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 3187     3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

3188 4. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP1}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

3189 The internal matrices and masks are checked for shape compatibility. The following conditions  
3190 must hold:

3191 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$ .

3192 2.  $\text{size}(\tilde{\mathbf{w}}) = \text{ncols}(\tilde{\mathbf{A}})$ .

3193 3.  $\text{size}(\tilde{\mathbf{u}}) = \text{nrows}(\tilde{\mathbf{A}})$ .

3194 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the dimension mismatch  
3195 error listed above is returned.

3196 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
3197 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3198 We are now ready to carry out the vector-matrix multiplication and any additional associated  
3199 operations. We describe this in terms of two intermediate vectors:

- 3200 •  $\tilde{\mathbf{t}}$ : The vector holding the product of vector  $\tilde{\mathbf{u}}^T$  and matrix  $\tilde{\mathbf{A}}$ .
- 3201 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3202 The intermediate vector  $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{ncols}(\tilde{\mathbf{A}}), \{(j, t_j) : \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{A}}(:, j)) \neq \emptyset\} \rangle$  is created.  
3203 The value of each of its elements is computed by

$$3204 \quad t_j = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{A}}(:, j))} (\tilde{\mathbf{u}}(k) \otimes \tilde{\mathbf{A}}(k, j)),$$

3205 where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring `op`, respectively.

3206 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3207 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 3208 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$3209 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3210 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
3211 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 3212 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 3213 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3214 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3215 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3216 \end{aligned}$$

3217 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3218 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 3219 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3220 mask which acts as a “write mask”.

- 3221 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are  
 3222 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$3223 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3224 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3225 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 3226 mask are unchanged:

$$3227 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3228 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 3229 of vector  $\mathbf{w}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 3230 exits with return value GrB\_SUCCESS and the new content of vector  $\mathbf{w}$  is as defined above but  
 3231 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3232 sequence.

### 3233 4.3.3 mxv: Matrix-vector multiply

3234 Multiplies a matrix by a vector on a semiring. The result is a vector.

## 3235 C Syntax

```
3236      GrB_Info GrB_mxv(GrB_Vector      w,
3237                      const GrB_Vector mask,
3238                      const GrB_BinaryOp accum,
3239                      const GrB_Semiring op,
3240                      const GrB_Matrix A,
3241                      const GrB_Vector u,
3242                      const GrB_Descriptor desc);
```

## 3243 Parameters

3244 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
 3245 that may be accumulated with the result of the matrix-vector product. On output,  
 3246 this vector holds the results of the operation.

3247 **mask** (IN) An optional “write” mask that controls which results from this operation are  
 3248 stored into the output vector  $\mathbf{w}$ . The mask dimensions must match those of the  
 3249 vector  $\mathbf{w}$ . If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain

3250 of the `mask` vector must be of type `bool` or any of the predefined “built-in” types  
 3251 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
 3252 dimensions of `w`), `GrB_NULL` should be specified.

3253 `accum` (IN) An optional binary operator used for accumulating entries into existing `w`  
 3254 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
 3255 specified.

3256 `op` (IN) Semiring used in the vector-matrix multiply.

3257 `A` (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
 3258 multiplication.

3259 `u` (IN) The GraphBLAS vector holding the values for the right-hand vector in the  
 3260 multiplication.

3261 `desc` (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
 3262 should be specified. Non-default field/value pairs are listed as follows:  
 3263

Param	Field	Value	Description
<code>w</code>	<code>GrB_OUTP</code>	<code>GrB_REPLACE</code>	Output vector <code>w</code> is cleared (all elements removed) before the result is stored in it.
<code>mask</code>	<code>GrB_MASK</code>	<code>GrB_STRUCTURE</code>	The write mask is constructed from the structure (pattern of stored values) of the input <code>mask</code> vector. The stored values are not examined.
<code>mask</code>	<code>GrB_MASK</code>	<code>GrB_COMP</code>	Use the complement of <code>mask</code> .
<code>A</code>	<code>GrB_INP0</code>	<code>GrB_TRAN</code>	Use transpose of <code>A</code> for the operation.

## 3265 Return Values

3266 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
 3267 blocking mode, this indicates that the compatibility tests on di-  
 3268 mensions and domains for the input arguments passed successfully.  
 3269 Either way, output vector `w` is ready to be used in the next method  
 3270 of the sequence.

3271 `GrB_PANIC` Unknown internal error.

3272 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque  
 3273 GraphBLAS objects (input or output) is in an invalid state caused  
 3274 by a previous execution error. Call `GrB_error()` to access any error  
 3275 messages generated by the implementation.

3276 `GrB_OUT_OF_MEMORY` Not enough memory available for the operation.

3277 `GrB_UNINITIALIZED_OBJECT` One or more of the GraphBLAS objects has not been initialized by  
 3278 a call to `new` (or `dup` for matrix or vector parameters).

3279 GrB\_DIMENSION\_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

3280 GrB\_DOMAIN\_MISMATCH The domains of the various vectors/matrices are incompatible with  
3281 the corresponding domains of the semiring or accumulation opera-  
3282 tor, or the mask's domain is not compatible with **bool** (in the case  
3283 where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

## 3284 Description

3285 GrB\_mvx computes the matrix-vector product  $w = A \oplus . \otimes u$ , or, if an optional binary accumulation  
3286 operator ( $\odot$ ) is provided,  $w = w \odot (A \oplus . \otimes u)$  (where matrix  $A$  can be optionally transposed).  
3287 Logically, this operation occurs in three steps:

3288 **Setup** The internal vectors, matrices and mask used in the computation are formed and their  
3289 domains/dimensions are tested for compatibility.

3290 **Compute** The indicated computations are carried out.

3291 **Output** The result is written into the output vector, possibly under control of a mask.

3292 Up to four argument vectors or matrices are used in the GrB\_mvx operation:

- 3293 1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3294 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3295 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3296 4.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

3297 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are  
3298 tested for domain compatibility as follows:

- 3299 1. If **mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{mask})$   
3300 must be from one of the pre-defined types of Table 3.2.
- 3301 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the semiring.
- 3302 3.  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the semiring.
- 3303 4.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the semiring.
- 3304 5. If **accum** is not GrB\_NULL, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
3305 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
3306 of the accumulation operator.

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

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

1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .
  - (b) If `mask  $\neq$  GrB_NULL`,
    - i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
    - ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
3. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
4. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

The internal matrices and masks are checked for shape compatibility. The following conditions must hold:

1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$ .
2.  $\text{size}(\tilde{\mathbf{w}}) = \text{nrows}(\tilde{\mathbf{A}})$ .
3.  $\text{size}(\tilde{\mathbf{u}}) = \text{ncols}(\tilde{\mathbf{A}})$ .

If any compatibility rule above is violated, execution of `GrB_m xv` ends and the dimension mismatch error listed above is returned.

From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

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

- $\tilde{\mathbf{t}}$ : The vector holding the product of matrix  $\tilde{\mathbf{A}}$  and vector  $\tilde{\mathbf{u}}$ .
- $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector  $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{nrows}(\tilde{\mathbf{A}}), \{(i, t_i) : \text{ind}(\tilde{\mathbf{A}}(i, :)) \cap \text{ind}(\tilde{\mathbf{u}}) \neq \emptyset\} \rangle$  is created. The value of each of its elements is computed by

$$t_i = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{A}}(i, :)) \cap \text{ind}(\tilde{\mathbf{u}})} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{u}}(k)),$$

3339 where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring **op**, respectively.  
 3340 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3341 • If **accum** = **GrB\_NULL**, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 3342 • If **accum** is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$3343 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3344 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 3345 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 3346 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 3347 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3348 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3349 \end{aligned}$$

3350 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

3351 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector **w**,  
 3352 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3353 mask which acts as a “write mask”.

- 3354 • If **desc[GrB\_OUTP].GrB\_REPLACE** is set, then any values in **w** on input to this operation are  
 3355 deleted and the content of the new output vector, **w**, is defined as,

$$3356 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3357 • If **desc[GrB\_OUTP].GrB\_REPLACE** is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3358 copied into the result vector, **w**, and elements of **w** that fall outside the set indicated by the  
 3359 mask are unchanged:  
 3360

$$3361 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3362 In **GrB\_BLOCKING** mode, the method exits with return value **GrB\_SUCCESS** and the new content  
 3363 of vector **w** is as defined above and fully computed. In **GrB\_NONBLOCKING** mode, the method  
 3364 exits with return value **GrB\_SUCCESS** and the new content of vector **w** is as defined above but  
 3365 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3366 sequence.

#### 3367 4.3.4 eWiseMult: Element-wise multiplication

3368 **Note:** The difference between **eWiseAdd** and **eWiseMult** is not about the element-wise operation  
 3369 but how the index sets are treated. **eWiseAdd** returns an object whose indices are the “union” of  
 3370 the indices of the inputs whereas **eWiseMult** returns an object whose indices are the “intersection”  
 3371 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on  
 3372 the set of values from the resulting index set.

#### 3373 4.3.4.1 eWiseMult: Vector variant

3374 Perform element-wise (general) multiplication on the intersection of elements of two vectors, pro-  
3375 ducing a third vector as result.

#### 3376 C Syntax

```
3377     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3378                           const GrB_Vector mask,  
3379                           const GrB_BinaryOp accum,  
3380                           const GrB_Semiring op,  
3381                           const GrB_Vector u,  
3382                           const GrB_Vector v,  
3383                           const GrB_Descriptor desc);  
3384  
3385     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3386                           const GrB_Vector mask,  
3387                           const GrB_BinaryOp accum,  
3388                           const GrB_Monoid op,  
3389                           const GrB_Vector u,  
3390                           const GrB_Vector v,  
3391                           const GrB_Descriptor desc);  
3392  
3393     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3394                           const GrB_Vector mask,  
3395                           const GrB_BinaryOp accum,  
3396                           const GrB_BinaryOp op,  
3397                           const GrB_Vector u,  
3398                           const GrB_Vector v,  
3399                           const GrB_Descriptor desc);
```

#### 3400 Parameters

3401 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3402 that may be accumulated with the result of the element-wise operation. On output,  
3403 this vector holds the results of the operation.

3404 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3405 stored into the output vector **w**. The mask dimensions must match those of the  
3406 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3407 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3408 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
3409 dimensions of **w**), **GrB\_NULL** should be specified.

3410 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**



3411 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
 3412 specified.

3413 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”  
 3414 operation. Depending on which type is passed, the following defines the binary  
 3415 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

3416 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

3417 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-  
 3418 nored.

3419 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  
 3420 is ignored.

3421 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the  
 3422 operation.

3423 **v** (IN) The GraphBLAS vector holding the values for the right-hand vector in the  
 3424 operation.

3425 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
 3426 should be specified. Non-default field/value pairs are listed as follows:  
 3427

Param	Field	Value	Description
<b>w</b>	<code>GrB_OUTP</code>	<code>GrB_REPLACE</code>	Output vector <b>w</b> is cleared (all elements removed) before the result is stored in it.
<b>mask</b>	<code>GrB_MASK</code>	<code>GrB_STRUCTURE</code>	The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined.
<b>mask</b>	<code>GrB_MASK</code>	<code>GrB_COMP</code>	Use the complement of <b>mask</b> .

## 3429 Return Values

3430 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 3431 blocking mode, this indicates that the compatibility tests on di-  
 3432 mensions and domains for the input arguments passed successfully.  
 3433 Either way, output vector **w** is ready to be used in the next method  
 3434 of the sequence.

3435 **GrB\_PANIC** Unknown internal error.

3436 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 3437 GraphBLAS objects (input or output) is in an invalid state caused  
 3438 by a previous execution error. Call `GrB_error()` to access any error  
 3439 messages generated by the implementation.

3440 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

3441 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
3442 a call to `new` (or `dup` for vector parameters).

3443 GrB\_DIMENSION\_MISMATCH Mask or vector dimensions are incompatible.

3444 GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with the cor-  
3445 responding domains of the binary operator (`op`) or accumulation  
3446 operator, or the mask's domain is not compatible with `bool` (in the  
3447 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3448 Description

3449 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS vectors:  
3450  $\mathbf{w} = \mathbf{u} \otimes \mathbf{v}$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $\mathbf{w} = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$ .  
3451 Logically, this operation occurs in three steps:

3452 **Setup** The internal vectors and mask used in the computation are formed and their domains  
3453 and dimensions are tested for compatibility.

3454 **Compute** The indicated computations are carried out.

3455 **Output** The result is written into the output vector, possibly under control of a mask.

3456 Up to four argument vectors are used in the `GrB_eWiseMult` operation:

- 3457 1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3458 2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3459 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3460 4.  $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

3461 The argument vectors, the “product” operator (`op`), and the accumulation operator (if provided)  
3462 are tested for domain compatibility as follows:

- 3463 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
3464 must be from one of the pre-defined types of Table 3.2.
- 3465 2.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$ .
- 3466 3.  $\mathbf{D}(\mathbf{v})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$ .
- 3467 4.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$ .
- 3468 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
3469 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of  
3470 the accumulation operator.

3471 Two domains are compatible with each other if values from one domain can be cast to values in  
 3472 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 3473 compatible with each other. A domain from a user-defined type is only compatible with itself. If any  
 3474 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch  
 3475 error listed above is returned.

3476 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
 3477 denotes copy):

- 3478 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3479 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 3480 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .
  - 3481 (b) If `mask  $\neq$  GrB_NULL`,
    - 3482 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
    - 3483 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 3484 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 3485 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 3486 4. Vector  $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$ .

3487 The internal vectors and mask are checked for dimension compatibility. The following conditions  
 3488 must hold:

- 3489 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}}) = \text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{v}})$ .

3490 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension  
 3491 mismatch error listed above is returned.

3492 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3493 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3494 We are now ready to carry out the element-wise “product” and any additional associated operations.  
 3495 We describe this in terms of two intermediate vectors:

- 3496 •  $\tilde{\mathbf{t}}$ : The vector holding the element-wise “product” of  $\tilde{\mathbf{u}}$  and vector  $\tilde{\mathbf{v}}$ .
- 3497 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3498 The intermediate vector  $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, t_i) : \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}}) \neq \emptyset\} \rangle$  is created. The  
 3499 value of each of its elements is computed by:

$$3500 \quad t_i = (\tilde{\mathbf{u}}(i) \otimes \tilde{\mathbf{v}}(i)), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}}))$$

3501 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

3502 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .

3503 • If  $\text{accum}$  is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$3504 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3505 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 3506 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$3507 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

3508

$$3509 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3510

$$3511 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3512 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3513 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 3514 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3515 mask which acts as a “write mask”.

3516 • If  $\text{desc}[\text{GrB\_OUTP}].\text{GrB\_REPLACE}$  is set, then any values in  $\mathbf{w}$  on input to this operation are  
 3517 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$3518 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3519 • If  $\text{desc}[\text{GrB\_OUTP}].\text{GrB\_REPLACE}$  is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3520 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 3521 mask are unchanged:

$$3522 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3523 In **GrB\_BLOCKING** mode, the method exits with return value **GrB\_SUCCESS** and the new content  
 3524 of vector  $\mathbf{w}$  is as defined above and fully computed. In **GrB\_NONBLOCKING** mode, the method  
 3525 exits with return value **GrB\_SUCCESS** and the new content of vector  $\mathbf{w}$  is as defined above but  
 3526 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3527 sequence.

#### 3528 4.3.4.2 eWiseMult: Matrix variant

3529 Perform element-wise (general) multiplication on the intersection of elements of two matrices, pro-  
 3530 ducing a third matrix as result.

## 3531 C Syntax

```

3532     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3533                           const GrB_Matrix Mask,
3534                           const GrB_BinaryOp accum,
3535                           const GrB_Semiring op,
3536                           const GrB_Matrix A,
3537                           const GrB_Matrix B,
3538                           const GrB_Descriptor desc);
3539
3540     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3541                           const GrB_Matrix Mask,
3542                           const GrB_BinaryOp accum,
3543                           const GrB_Monoid op,
3544                           const GrB_Matrix A,
3545                           const GrB_Matrix B,
3546                           const GrB_Descriptor desc);
3547
3548     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3549                           const GrB_Matrix Mask,
3550                           const GrB_BinaryOp accum,
3551                           const GrB_BinaryOp op,
3552                           const GrB_Matrix A,
3553                           const GrB_Matrix B,
3554                           const GrB_Descriptor desc);

```

## 3555 Parameters

3556 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
3557 that may be accumulated with the result of the element-wise operation. On output,  
3558 the matrix holds the results of the operation.

3559 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
3560 stored into the output matrix C. The mask dimensions must match those of the  
3561 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
3562 of the **Mask** matrix must be of type `bool` or any of the predefined “built-in” types  
3563 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
3564 dimensions of C), `GrB_NULL` should be specified.

3565 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
3566 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
3567 specified.

3568 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”  
3569 operation. Depending on which type is passed, the following defines the binary  
3570 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

3571  
3572  
3573  
3574  
3575  
  
3576  
3577  
  
3578  
3579  
  
3580  
3581  
3582  
  
  
3583  
  
  
  
  
  
  
  
  
  
3584  
  
  
3585  
3586  
3587  
3588  
3589  
  
3590  
  
3591  
3592  
3593  
3594  
  
3595  
  
3596  
3597  
  
3598

BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .  
 Monoid:  $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$ ; the identity element is ignored.  
 Semiring:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes(\text{op}) \rangle$ ; the additive monoid is ignored.

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

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

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

Param	Field	Value	Description
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.

3599 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
 3600 corresponding domains of the binary operator ( $\otimes$ ) or accumulation  
 3601 operator, or the mask's domain is not compatible with `bool` (in the  
 3602 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3603 Description

3604 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS matrices:  
 3605  $C = A \otimes B$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot (A \otimes B)$ .  
 3606 Logically, this operation occurs in three steps:

3607 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 3608 and dimensions are tested for compatibility.

3609 **Compute** The indicated computations are carried out.

3610 **Output** The result is written into the output matrix, possibly under control of a mask.

3611 Up to four argument matrices are used in the `GrB_eWiseMult` operation:

- 3612 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3613 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)
- 3614 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3615 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3616 The argument matrices, the “product” operator ( $\otimes$ ), and the accumulation operator (if provided)  
 3617 are tested for domain compatibility as follows:

- 3618 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 3619 must be from one of the pre-defined types of Table 3.2.
- 3620 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\otimes)$ .
- 3621 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\otimes)$ .
- 3622 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\otimes)$ .
- 3623 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 3624 of the accumulation operator and  $\mathbf{D}_{out}(\otimes)$  of  $\otimes$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
 3625 the accumulation operator.

3626 Two domains are compatible with each other if values from one domain can be cast to values in  
 3627 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 3628 compatible with each other. A domain from a user-defined type is only compatible with itself. If any

3629 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch  
 3630 error listed above is returned.

3631 From the argument matrices, the internal matrices and mask used in the computation are formed  
 3632 ( $\leftarrow$  denotes copy):

- 3633 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 3634 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
  - 3635 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 3636  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 3637 (b) If `Mask  $\neq$  GrB_NULL`,
    - 3638 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
 3639  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 3640 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
 3641  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\text{bool})\mathbf{Mask}(i, j) = \text{true}\} \rangle$ .
  - 3642 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 3643 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 3644 4. Matrix  $\tilde{\mathbf{B}} \leftarrow \text{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}$ .

3645 The internal matrices and masks are checked for dimension compatibility. The following conditions  
 3646 must hold:

- 3647 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$ .
- 3648 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$ .

3649 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension  
 3650 mismatch error listed above is returned.

3651 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3652 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3653 We are now ready to carry out the element-wise “product” and any additional associated operations.  
 3654 We describe this in terms of two intermediate matrices:

- 3655 •  $\tilde{\mathbf{T}}$ : The matrix holding the element-wise product of  $\tilde{\mathbf{A}}$  and  $\tilde{\mathbf{B}}$ .
- 3656 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

3657 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\} \rangle$   
 3658 is created. The value of each of its elements is computed by

$$3659 \quad T_{ij} = (\tilde{\mathbf{A}}(i, j) \otimes \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})$$

3660 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:



3661 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .

3662 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$3663 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3664 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
3665 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$3666 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$3667 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$3669 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3671 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3672 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
3673 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
3674 mask which acts as a “write mask”.

3675 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
3676 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$3677 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3678 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
3679 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
3680 mask are unchanged:

$$3681 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3682 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
3683 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
3684 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
3685 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
3686 sequence.

#### 3687 4.3.5 eWiseAdd: Element-wise addition

3688 **Note:** The difference between `eWiseAdd` and `eWiseMult` is not about the element-wise operation  
3689 but how the index sets are treated. `eWiseAdd` returns an object whose indices are the “union” of  
3690 the indices of the inputs whereas `eWiseMult` returns an object whose indices are the “intersection”  
3691 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on  
3692 the set of values from the resulting index set.

#### 3693 4.3.5.1 eWiseAdd: Vector variant

3694 Perform element-wise (general) addition on the elements of two vectors, producing a third vector  
3695 as result.

#### 3696 C Syntax

```
3697     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3698                           const GrB_Vector mask,  
3699                           const GrB_BinaryOp accum,  
3700                           const GrB_Semiring op,  
3701                           const GrB_Vector u,  
3702                           const GrB_Vector v,  
3703                           const GrB_Descriptor desc);  
3704  
3705     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3706                           const GrB_Vector mask,  
3707                           const GrB_BinaryOp accum,  
3708                           const GrB_Monoid op,  
3709                           const GrB_Vector u,  
3710                           const GrB_Vector v,  
3711                           const GrB_Descriptor desc);  
3712  
3713     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3714                           const GrB_Vector mask,  
3715                           const GrB_BinaryOp accum,  
3716                           const GrB_BinaryOp op,  
3717                           const GrB_Vector u,  
3718                           const GrB_Vector v,  
3719                           const GrB_Descriptor desc);
```

#### 3720 Parameters

3721 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3722 that may be accumulated with the result of the element-wise operation. On output,  
3723 this vector holds the results of the operation.

3724 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3725 stored into the output vector **w**. The mask dimensions must match those of the  
3726 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3727 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3728 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
3729 dimensions of **w**), **GrB\_NULL** should be specified.

3730 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**

3731 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
 3732 specified.

3733 op (IN) The semiring, monoid, or binary operator used in the element-wise “sum”  
 3734 operation. Depending on which type is passed, the following defines the binary  
 3735 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$ , used:

3736 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

3737 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-  
 3738 nored.

3739 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-  
 3740 nary op and additive identity are ignored.

3741 u (IN) The GraphBLAS vector holding the values for the left-hand vector in the  
 3742 operation.

3743 v (IN) The GraphBLAS vector holding the values for the right-hand vector in the  
 3744 operation.

3745 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 3746 should be specified. Non-default field/value pairs are listed as follows:  
 3747

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.

## 3749 Return Values

3750 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 3751 blocking mode, this indicates that the compatibility tests on di-  
 3752 mensions and domains for the input arguments passed successfully.  
 3753 Either way, output vector w is ready to be used in the next method  
 3754 of the sequence.

3755 GrB\_PANIC Unknown internal error.

3756 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 3757 GraphBLAS objects (input or output) is in an invalid state caused  
 3758 by a previous execution error. Call GrB\_error() to access any error  
 3759 messages generated by the implementation.

3760 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

3761 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
 3762 a call to `new` (or `dup` for vector parameters).

3763 GrB\_DIMENSION\_MISMATCH Mask or vector dimensions are incompatible.

3764 GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with the cor-  
 3765 responding domains of the binary operator (`op`) or accumulation  
 3766 operator, or the mask's domain is not compatible with `bool` (in the  
 3767 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3768 Description

3769 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS vectors:  $w =$   
 3770  $u \oplus v$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $w = w \odot (u \oplus v)$ . Logically,  
 3771 this operation occurs in three steps:

3772 **Setup** The internal vectors and mask used in the computation are formed and their domains  
 3773 and dimensions are tested for compatibility.

3774 **Compute** The indicated computations are carried out.

3775 **Output** The result is written into the output vector, possibly under control of a mask.

3776 Up to four argument vectors are used in the `GrB_eWiseAdd` operation:

- 3777 1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3778 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3779 3.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$
- 3780 4.  $v = \langle \mathbf{D}(v), \mathbf{size}(v), \mathbf{L}(v) = \{(i, v_i)\} \rangle$

3781 The argument vectors, the “sum” operator (`op`), and the accumulation operator (if provided) are  
 3782 tested for domain compatibility as follows:

- 3783 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
 3784 must be from one of the pre-defined types of Table 3.2.
- 3785 2.  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$ .
- 3786 3.  $\mathbf{D}(v)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$ .
- 3787 4.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3788 5.  $\mathbf{D}(u)$  and  $\mathbf{D}(v)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3789 6. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 3790 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
 3791 the accumulation operator.

3792 Two domains are compatible with each other if values from one domain can be cast to values in  
 3793 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 3794 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 3795 any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the domain mismatch  
 3796 error listed above is returned.

3797 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
 3798 denotes copy):

- 3799 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3800 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 3801 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .
  - 3802 (b) If `mask  $\neq$  GrB_NULL`,
    - 3803 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
    - 3804 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 3805 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 3806 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 3807 4. Vector  $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$ .

3808 The internal vectors and mask are checked for dimension compatibility. The following conditions  
 3809 must hold:

- 3810 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}}) = \text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{v}})$ .

3811 If any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the dimension  
 3812 mismatch error listed above is returned.

3813 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3814 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3815 We are now ready to carry out the element-wise “sum” and any additional associated operations.  
 3816 We describe this in terms of two intermediate vectors:

- 3817 •  $\tilde{\mathbf{t}}$ : The vector holding the element-wise “sum” of  $\tilde{\mathbf{u}}$  and vector  $\tilde{\mathbf{v}}$ .
- 3818 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3819 The intermediate vector  $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, t_i) : \text{ind}(\tilde{\mathbf{u}}) \cup \text{ind}(\tilde{\mathbf{v}}) \neq \emptyset\} \rangle$  is created. The  
 3820 value of each of its elements is computed by:

$$\begin{aligned}
 3821 \quad t_i &= (\tilde{\mathbf{u}}(i) \oplus \tilde{\mathbf{v}}(i)), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}})) \\
 3822 \\
 3823 \quad t_i &= \tilde{\mathbf{u}}(i), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) - (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}})))
 \end{aligned}$$

3824  
3825

$$t_i = \tilde{\mathbf{v}}(i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{v}}) - (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}})))$$

3826

where the difference operator in the previous expressions refers to set difference.

3827

The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

3828

- If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .

3829

- If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

3830

$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3831

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  $\tilde{\mathbf{t}}$ .

3832

3833

$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

3834

$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

3835

3836

$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

3837

3838

where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3839

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”.

3840

3841

3842

- 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,

3843

3844

$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3845

- 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:

3846

3847

3848

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3849

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.

3850

3851

3852

3853

3854

#### 4.3.5.2 eWiseAdd: Matrix variant

3855

Perform element-wise (general) addition on the elements of two matrices, producing a third matrix as result.

3856

## 3857 C Syntax

```

3858     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3859                          const GrB_Matrix Mask,
3860                          const GrB_BinaryOp accum,
3861                          const GrB_Semiring op,
3862                          const GrB_Matrix A,
3863                          const GrB_Matrix B,
3864                          const GrB_Descriptor desc);
3865
3866     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3867                          const GrB_Matrix Mask,
3868                          const GrB_BinaryOp accum,
3869                          const GrB_Monoid op,
3870                          const GrB_Matrix A,
3871                          const GrB_Matrix B,
3872                          const GrB_Descriptor desc);
3873
3874     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3875                          const GrB_Matrix Mask,
3876                          const GrB_BinaryOp accum,
3877                          const GrB_BinaryOp op,
3878                          const GrB_Matrix A,
3879                          const GrB_Matrix B,
3880                          const GrB_Descriptor desc);

```

## 3881 Parameters

3882 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
3883 that may be accumulated with the result of the element-wise operation. On output,  
3884 the matrix holds the results of the operation.

3885 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
3886 stored into the output matrix C. The mask dimensions must match those of the  
3887 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
3888 of the **Mask** matrix must be of type `bool` or any of the predefined “built-in” types  
3889 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
3890 dimensions of C), `GrB_NULL` should be specified.

3891 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
3892 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
3893 specified.

3894 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”  
3895 operation. Depending on which type is passed, the following defines the binary  
3896 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$ , used:

3897 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .  
 3898 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-  
 3899 nored.  
 3900 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-  
 3901 nary op and additive identity are ignored.

3902 A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
 3903 operation.

3904 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
 3905 operation.

3906 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 3907 should be specified. Non-default field/value pairs are listed as follows:  
 3908

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.

## 3910 Return Values

3911 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 3912 blocking mode, this indicates that the compatibility tests on di-  
 3913 mensions and domains for the input arguments passed successfully.  
 3914 Either way, output matrix C is ready to be used in the next method  
 3915 of the sequence.

3916 GrB\_PANIC Unknown internal error.

3917 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 3918 GraphBLAS objects (input or output) is in an invalid state caused  
 3919 by a previous execution error. Call GrB\_error() to access any error  
 3920 messages generated by the implementation.

3921 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

3922 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
 3923 a call to new (or Matrix\_dup for matrix parameters).

3924 GrB\_DIMENSION\_MISMATCH Mask and/or matrix dimensions are incompatible.



3925 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
 3926 corresponding domains of the binary operator ( $\text{op}$ ) or accumulation  
 3927 operator, or the mask's domain is not compatible with `bool` (in the  
 3928 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3929 Description

3930 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS matrices:  
 3931  $C = A \oplus B$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot (A \oplus B)$ .  
 3932 Logically, this operation occurs in three steps:

3933 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 3934 and dimensions are tested for compatibility.

3935 **Compute** The indicated computations are carried out.

3936 **Output** The result is written into the output matrix, possibly under control of a mask.

3937 Up to four argument matrices are used in the `GrB_eWiseAdd` operation:

- 3938 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3939 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)
- 3940 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3941 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3942 The argument matrices, the “sum” operator ( $\text{op}$ ), and the accumulation operator (if provided) are  
 3943 tested for domain compatibility as follows:

- 3944 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 3945 must be from one of the pre-defined types of Table 3.2.
- 3946 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$ .
- 3947 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$ .
- 3948 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3949 5.  $\mathbf{D}(A)$  and  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3950 6. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 3951 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of  $\text{op}$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
 3952 the accumulation operator.

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

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

1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
  - (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - (b) If `Mask  $\neq$  GrB_NULL`,
    - i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
3. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
4. Matrix  $\tilde{\mathbf{B}} \leftarrow \text{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}$ .

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

1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$ .
2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$ .

If any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the dimension mismatch error listed above is returned.

From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

We are now ready to carry out the element-wise “sum” and any additional associated operations. We describe this in terms of two intermediate matrices:

- $\tilde{\mathbf{T}}$ : The matrix holding the element-wise sum of  $\tilde{\mathbf{A}}$  and  $\tilde{\mathbf{B}}$ .
- $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

3984 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cup \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\}$   
 3985 is created. The value of each of its elements is computed by

$$\begin{aligned}
 3986 \quad T_{ij} &= (\tilde{\mathbf{A}}(i, j) \oplus \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \\
 3987 \quad T_{ij} &= \tilde{\mathbf{A}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{A}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}))) \\
 3988 \quad T_{ij} &= \tilde{\mathbf{B}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{B}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}))) \\
 3990
 \end{aligned}$$

3991 where the difference operator in the previous expressions refers to set difference.

3992 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 3993 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 3994 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$3995 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3996 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 3997 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned}
 3998 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\
 3999 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 4000 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 4001 \\
 4002
 \end{aligned}$$

4003 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4004 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 4005 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 4006 mask which acts as a “write mask”.

- 4007 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
 4008 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$4009 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 4010 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 4011 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 4012 mask are unchanged:

$$4013 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4014 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
 4015 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
 4016 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
 4017 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 4018 sequence.

### 4019 4.3.6 extract: Selecting sub-graphs

4020 Extract a subset of a matrix or vector.

#### 4021 4.3.6.1 extract: Standard vector variant

4022 Extract a sub-vector from a larger vector as specified by a set of indices. The result is a vector  
4023 whose size is equal to the number of indices.

### 4024 C Syntax

```
4025         GrB_Info GrB_extract(GrB_Vector          w,  
4026                             const GrB_Vector    mask,  
4027                             const GrB_BinaryOp   accum,  
4028                             const GrB_Vector    u,  
4029                             const GrB_Index     *indices,  
4030                             GrB_Index           nindices,  
4031                             const GrB_Descriptor desc);
```

### 4032 Parameters

4033 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
4034 that may be accumulated with the result of the extract operation. On output, this  
4035 vector holds the results of the operation.

4036 **mask** (IN) An optional “write” mask that controls which results from this operation are  
4037 stored into the output vector **w**. The mask dimensions must match those of the  
4038 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4039 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
4040 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4041 dimensions of **w**), **GrB\_NULL** should be specified.

4042 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
4043 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4044 specified.

4045 **u** (IN) The GraphBLAS vector from which the subset is extracted.

4046 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations of  
4047 elements from **u** that are extracted. If all elements of **u** are to be extracted in order  
4048 from 0 to **nindices** – 1, then **GrB\_ALL** should be specified. Regardless of execution  
4049 mode and return value, this array may be manipulated by the caller after this  
4050 operation returns without affecting any deferred computations for this operation.

4051 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(w)**.

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
w	GrB_OUTP	GrB_REPLACE	Output vector <b>w</b> is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of <b>mask</b> .

## Return Values

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

GrB\_PANIC Unknown internal error.

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

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

GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to **new** (or **dup** for vector parameters).

GrB\_INDEX\_OUT\_OF\_BOUNDS A value in **indices** is greater than or equal to **size(u)**. In non-blocking mode, this error can be deferred.

GrB\_DIMENSION\_MISMATCH **mask** and **w** dimensions are incompatible, or **nindices**  $\neq$  **size(w)**.

GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with **bool** (in the case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

GrB\_NULL\_POINTER Argument **row\_indices** is a NULL pointer.

## Description

This variant of GrB\_extract computes the result of extracting a subset of locations from a GraphBLAS vector in a specific order:  $w = u(\text{indices})$ ; or, if an optional binary accumulation operator

4081  $(\odot)$  is provided,  $w = w \odot u(\text{indices})$ . More explicitly:

$$4082 \quad \begin{aligned} w(i) &= u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ w(i) &= w(i) \odot u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices} \end{aligned}$$

4083 Logically, this operation occurs in three steps:

4084     **Setup** The internal vectors and mask used in the computation are formed and their domains  
4085             and dimensions are tested for compatibility.

4086     **Compute** The indicated computations are carried out.

4087     **Output** The result is written into the output vector, possibly under control of a mask.

4088 Up to three argument vectors are used in this `GrB_extract` operation:

- 4089     1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4090     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4091     3.  $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

4092 The argument vectors and the accumulation operator (if provided) are tested for domain compati-  
4093 bility as follows:

- 4094     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
4095         must be from one of the pre-defined types of Table 3.2.
- 4096     2.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}(u)$ .
- 4097     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
4098         of the accumulation operator and  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
4099         mulation operator.

4100 Two domains are compatible with each other if values from one domain can be cast to values in  
4101 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4102 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4103 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch  
4104 error listed above is returned.

4105 From the arguments, the internal vectors, mask, and index array used in the computation are  
4106 formed ( $\leftarrow$  denotes copy):

- 4107     1. Vector  $\tilde{w} \leftarrow w$ .
- 4108     2. One-dimensional mask,  $\tilde{m}$ , is computed from argument `mask` as follows:  
4109         (a) If `mask` = `GrB_NULL`, then  $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$ .

- 4110 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,  
 4111 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,  
 4112 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .  
 4113 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .
- 4114 3. Vector  $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 4115 4. The internal index array,  $\widetilde{\mathbf{I}}$ , is computed from argument indices as follows:
- 4116 (a) If  $\text{indices} = \text{GrB\_ALL}$ , then  $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$ .  
 4117 (b) Otherwise,  $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$ .

4118 The internal vectors and mask are checked for dimension compatibility. The following conditions  
 4119 must hold:

- 4120 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$   
 4121 2.  $\text{nindices} = \text{size}(\widetilde{\mathbf{w}})$ .

4122 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
 4123 match error listed above is returned.

4124 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 4125 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4126 We are now ready to carry out the extract and any additional associated operations. We describe  
 4127 this in terms of two intermediate vectors:

- 4128 •  $\widetilde{\mathbf{t}}$ : The vector holding the extraction from  $\widetilde{\mathbf{u}}$  in their destination locations relative to  $\widetilde{\mathbf{w}}$ .
- 4129 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4130 The intermediate vector,  $\widetilde{\mathbf{t}}$ , is created as follows:

$$4131 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{w}}), \{(i, \widetilde{\mathbf{u}}[\widetilde{\mathbf{I}}[i]]) \mid \forall i, 0 \leq i < \text{nindices} : \widetilde{\mathbf{I}}[i] \in \text{ind}(\widetilde{\mathbf{u}})\} \rangle.$$

4132 At this point, if any value in  $\widetilde{\mathbf{I}}$  is not in the valid range of indices for vector  $\widetilde{\mathbf{u}}$ , the execution of  
 4133 `GrB_extract` ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING`  
 4134 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
 4135 result vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4136 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 4137 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$ .
- 4138 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{z}}$  is defined as

$$4139 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ , using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a “write mask”.

- If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

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

#### 4.3.6.2 extract: Standard matrix variant

Extract a sub-matrix from a larger matrix as specified by a set of row indices and a set of column indices. The result is a matrix whose size is equal to size of the sets of indices.

### C Syntax

```
GrB_Info GrB_extract(GrB_Matrix      C,
                    const GrB_Matrix  Mask,
                    const GrB_BinaryOp accum,
                    const GrB_Matrix  A,
                    const GrB_Index   *row_indices,
                    GrB_Index          nrows,
                    const GrB_Index   *col_indices,
                    GrB_Index          ncols,
                    const GrB_Descriptor desc);
```



## Parameters

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

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

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

**A** (IN) The GraphBLAS matrix from which the subset is extracted.

**row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **A** from which elements are extracted. If elements in all rows of **A** are to be extracted in order, **GrB\_ALL** should be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation.

**nrows** (IN) The number of values in the **row\_indices** array. Must be equal to **nrows(C)**.

**col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns of **A** from which elements are extracted. If elements in all columns of **A** are to be extracted in order, then **GrB\_ALL** should be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation.

**ncols** (IN) The number of values in the **col\_indices** array. Must be equal to **ncols(C)**.

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

Param	Field	Value	Description
<b>C</b>	<b>GrB_OUTP</b>	<b>GrB_REPLACE</b>	Output matrix <b>C</b> is cleared (all elements removed) before the result is stored in it.
<b>Mask</b>	<b>GrB_MASK</b>	<b>GrB_STRUCTURE</b>	The write mask is constructed from the structure (pattern of stored values) of the input <b>Mask</b> matrix. The stored values are not examined.
<b>Mask</b>	<b>GrB_MASK</b>	<b>GrB_COMP</b>	Use the complement of <b>Mask</b> .
<b>A</b>	<b>GrB_INP0</b>	<b>GrB_TRAN</b>	Use transpose of <b>A</b> for the operation.

## 4206 Return Values

4207	<b>GrB_SUCCESS</b>	In blocking mode, the operation completed successfully. In non-
4208		blocking mode, this indicates that the compatibility tests on
4209		dimensions and domains for the input arguments passed suc-
4210		cessfully. Either way, output matrix C is ready to be used in the
4211		next method of the sequence.
4212	<b>GrB_PANIC</b>	Unknown internal error.
4213	<b>GrB_INVALID_OBJECT</b>	This is returned in any execution mode whenever one of the
4214		opaque GraphBLAS objects (input or output) is in an invalid
4215		state caused by a previous execution error. Call <code>GrB_error()</code> to
4216		access any error messages generated by the implementation.
4217	<b>GrB_OUT_OF_MEMORY</b>	Not enough memory available for the operation.
4218	<b>GrB_UNINITIALIZED_OBJECT</b>	One or more of the GraphBLAS objects has not been initialized
4219		by a call to <code>new</code> (or <code>Matrix_dup</code> for matrix parameters).
4220	<b>GrB_INDEX_OUT_OF_BOUNDS</b>	A value in <code>row_indices</code> is greater than or equal to <code>nrows(A)</code> , or
4221		a value in <code>col_indices</code> is greater than or equal to <code>ncols(A)</code> . In
4222		non-blocking mode, this error can be deferred.
4223	<b>GrB_DIMENSION_MISMATCH</b>	Mask and C dimensions are incompatible, <code>nrows</code> $\neq$ <code>nrows(C)</code> , or
4224		<code>ncols</code> $\neq$ <code>ncols(C)</code> .
4225	<b>GrB_DOMAIN_MISMATCH</b>	The domains of the various matrices are incompatible with each
4226		other or the corresponding domains of the accumulation oper-
4227		ator, or the mask's domain is not compatible with <code>bool</code> (in the
4228		case where <code>desc[GrB_MASK].GrB_STRUCTURE</code> is not set).
4229	<b>GrB_NULL_POINTER</b>	Either argument <code>row_indices</code> is a NULL pointer, argument <code>col_indices</code>
4230		is a NULL pointer, or both.

## 4231 Description

4232 This variant of `GrB_extract` computes the result of extracting a subset of locations from specified  
 4233 rows and columns of a GraphBLAS matrix in a specific order:  $C = A(\text{row\_indices}, \text{col\_indices})$ ; or,  
 4234 if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot A(\text{row\_indices}, \text{col\_indices})$ .  
 4235 More explicitly (not accounting for an optional transpose of A):

$$\begin{aligned}
 &C(i, j) = A(\text{row\_indices}[i], \text{col\_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\
 &C(i, j) = C(i, j) \odot A(\text{row\_indices}[i], \text{col\_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}
 \end{aligned}$$

4237 Logically, this operation occurs in three steps:

4238 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 4239 and dimensions are tested for compatibility.

4240 **Compute** The indicated computations are carried out.

4241 **Output** The result is written into the output matrix, possibly under control of a mask.

4242 Up to three argument matrices are used in the `GrB_extract` operation:

- 4243 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4244 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)
- 4245 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4246 The argument matrices and the accumulation operator (if provided) are tested for domain compat-  
4247 ibility as follows:

- 4248 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
4249 must be from one of the pre-defined types of Table 3.2.
- 4250 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$ .
- 4251 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
4252 of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
4253 mulation operator.

4254 Two domains are compatible with each other if values from one domain can be cast to values in  
4255 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4256 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4257 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch  
4258 error listed above is returned.

4259 From the arguments, the internal matrices, `mask`, and index arrays used in the computation are  
4260 formed ( $\leftarrow$  denotes copy):

- 4261 1. Matrix  $\tilde{C} \leftarrow C$ .
- 4262 2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument `Mask` as follows:
  - 4263 (a) If `Mask` = `GrB_NULL`, then  $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$   
4264  $j < \mathbf{ncols}(C)\} \rangle$ .
  - 4265 (b) If `Mask`  $\neq$  `GrB_NULL`,
    - 4266 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
4267  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
    - 4268 ii. Otherwise,  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
4269  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
  - 4270 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{M} \leftarrow \neg \tilde{M}$ .
- 4271 3. Matrix  $\tilde{A} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? A^T : A$ .

- 4272 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
- 4273 (a) If `row_indices` = `GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$ .
- 4274 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{row\_indices}[i], \forall i : 0 \leq i < \text{nrows}$ .
- 4275 5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:
- 4276 (a) If `col_indices` = `GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \text{ncols}$ .
- 4277 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \text{col\_indices}[j], \forall j : 0 \leq j < \text{ncols}$ .

4278 The internal matrices and mask are checked for dimension compatibility. The following conditions  
4279 must hold:

- 4280 1.  $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}(\tilde{\mathbf{M}})$ .
- 4281 2.  $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}(\tilde{\mathbf{M}})$ .
- 4282 3.  $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}$ .
- 4283 4.  $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}$ .

4284 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
4285 match error listed above is returned.

4286 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4287 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4288 We are now ready to carry out the extract and any additional associated operations. We describe  
4289 this in terms of two intermediate matrices:

- 4290 •  $\tilde{\mathbf{T}}$ : The matrix holding the extraction from  $\tilde{\mathbf{A}}$ .
- 4291 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

4292 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

4293 
$$\tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \{ (i, j, \tilde{\mathbf{A}}(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j])) \mid \forall (i, j), 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} : (\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j]) \in \text{ind}(\tilde{\mathbf{A}}) \} \rangle.$$

4294 At this point, if any value in the  $\tilde{\mathbf{I}}$  array is not in the range  $[0, \text{nrows}(\tilde{\mathbf{A}}))$  or any value in the  $\tilde{\mathbf{J}}$   
4295 array is not in the range  $[0, \text{ncols}(\tilde{\mathbf{A}}))$ , the execution of `GrB_extract` ends and the index out-of-  
4296 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred  
4297 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from  
4298 this point forward in the sequence.

4299 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 4300 • If `accum` = `GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .

4301 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$4302 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4303 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
4304 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$4305 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$4306 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$4308 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4310 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4311 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
4312 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
4313 mask which acts as a “write mask”.

4314 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
4315 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$4316 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4317 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
4318 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
4319 mask are unchanged:

$$4320 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4321 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
4322 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
4323 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
4324 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
4325 sequence.

#### 4326 4.3.6.3 extract: Column (and row) variant

4327 Extract from one column of a matrix into a vector. Note that with the transpose descriptor for the  
4328 source matrix, elements of an arbitrary row of the matrix can be extracted with this function as  
4329 well.

## 4330 C Syntax

```

4331     GrB_Info GrB_extract(GrB_Vector      w,
4332                          const GrB_Vector mask,
4333                          const GrB_BinaryOp accum,
4334                          const GrB_Matrix A,
4335                          const GrB_Index *row_indices,
4336                          GrB_Index nrows,
4337                          GrB_Index col_index,
4338                          const GrB_Descriptor desc);

```

## 4339 Parameters

4340     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
4341     that may be accumulated with the result of the extract operation. On output, this  
4342     vector holds the results of the operation.

4343     **mask** (IN) An optional “write” mask that controls which results from this operation are  
4344     stored into the output vector **w**. The mask dimensions must match those of the  
4345     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4346     of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
4347     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4348     dimensions of **w**), **GrB\_NULL** should be specified.

4349     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
4350     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4351     specified.

4352     **A** (IN) The GraphBLAS matrix from which the column subset is extracted.

4353     **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations  
4354     within the specified column of **A** from which elements are extracted. If elements in  
4355     all rows of **A** are to be extracted in order, **GrB\_ALL** should be specified. Regardless  
4356     of execution mode and return value, this array may be manipulated by the caller  
4357     after this operation returns without affecting any deferred computations for this  
4358     operation.

4359     **nrows** (IN) The number of indices in the **row\_indices** array. Must be equal to **size(w)**.

4360     **col\_index** (IN) The index of the column of **A** from which to extract values. It must be in the  
4361     range  $[0, \mathbf{ncols}(A))$ .

4362     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
4363     should be specified. Non-default field/value pairs are listed as follows:

4364

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector <b>w</b> is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of <b>mask</b> .
A	GrB_INP0	GrB_TRAN	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.

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

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

**GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by a call to **new** (or **dup** for vector or matrix parameters).

**GrB\_INVALID\_INDEX** **col\_index** is outside the allowable range (i.e., greater than **ncols(A)**).

**GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **row\_indices** is greater than or equal to **nrows(A)**. In non-blocking mode, this error can be deferred.

**GrB\_DIMENSION\_MISMATCH** **mask** and **w** dimensions are incompatible, or **nrows**  $\neq$  **size(w)**.

**GrB\_DOMAIN\_MISMATCH** The domains of the vector or matrix are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with **bool** (in the case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

**GrB\_NULL\_POINTER** Argument **row\_indices** is a NULL pointer.

## Description

This variant of **GrB\_extract** computes the result of extracting a subset of locations (in a specific order) from a specified column of a GraphBLAS matrix: **w** = **A(:, col\_index)(row\_indices)**; or, if

4392 an optional binary accumulation operator ( $\odot$ ) is provided,  $w = w \odot A(:, \text{col\_index})(\text{row\_indices})$ .  
 4393 More explicitly:

$$4394 \quad \begin{aligned} w(i) &= A(\text{row\_indices}[i], \text{col\_index}) \quad \forall i : 0 \leq i < \text{nrows}, \quad \text{or} \\ w(i) &= w(i) \odot A(\text{row\_indices}[i], \text{col\_index}) \quad \forall i : 0 \leq i < \text{nrows} \end{aligned}$$

4395 Logically, this operation occurs in three steps:

4396     **Setup** The internal matrices, vectors, and mask used in the computation are formed and their  
 4397 domains and dimensions are tested for compatibility.

4398     **Compute** The indicated computations are carried out.

4399     **Output** The result is written into the output vector, possibly under control of a mask.

4400 Up to three argument vectors and matrices are used in this GrB\_extract operation:

- 4401 1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4402 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4403 3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4404 The argument vectors, matrix and the accumulation operator (if provided) are tested for domain  
 4405 compatibility as follows:

- 4406 1. If **mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{mask})$   
 4407 must be from one of the pre-defined types of Table 3.2.
- 4408 2.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}(A)$ .
- 4409 3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 4410 of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 4411 mulation operator.

4412 Two domains are compatible with each other if values from one domain can be cast to values in  
 4413 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 4414 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 4415 any compatibility rule above is violated, execution of GrB\_extract ends and the domain mismatch  
 4416 error listed above is returned.

4417 From the arguments, the internal vector, matrix, mask, and index array used in the computation  
 4418 are formed ( $\leftarrow$  denotes copy):

- 4419 1. Vector  $\tilde{w} \leftarrow w$ .
- 4420 2. One-dimensional mask,  $\tilde{m}$ , is computed from argument **mask** as follows:  
 4421 (a) If **mask** = GrB\_NULL, then  $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$ .



- 4422 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,
- 4423 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
- 4424 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
- 4425 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .
- 4426 3. Matrix  $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 4427 4. The internal row index array,  $\widetilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
- 4428 (a) If `indices = GrB_ALL`, then  $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$ .
- 4429 (b) Otherwise,  $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nrows}$ .

4430 The internal vector, `mask`, and index array are checked for dimension compatibility. The following  
 4431 conditions must hold:

- 4432 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 4433 2.  $\text{size}(\widetilde{\mathbf{w}}) = \text{nrows}$ .

4434 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
 4435 match error listed above is returned.

4436 The `col_index` parameter is checked for a valid value. The following condition must hold:

- 4437 1.  $0 \leq \text{col\_index} < \text{ncols}(\mathbf{A})$

4438 If the rule above is violated, execution of `GrB_extract` ends and the invalid index error listed above  
 4439 is returned.

4440 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 4441 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4442 We are now ready to carry out the extract and any additional associated operations. We describe  
 4443 this in terms of two intermediate vectors:

- 4444 •  $\widetilde{\mathbf{t}}$ : The vector holding the extraction from a column of  $\widetilde{\mathbf{A}}$ .
- 4445 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4446 The intermediate vector,  $\widetilde{\mathbf{t}}$ , is created as follows:

4447 
$$\widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}, \{(i, \widetilde{\mathbf{A}}(\widetilde{\mathbf{I}}[i], \text{col\_index})) \mid \forall i, 0 \leq i < \text{nrows} : (\widetilde{\mathbf{I}}[i], \text{col\_index}) \in \text{ind}(\widetilde{\mathbf{A}})\} \rangle.$$

4448 At this point, if any value in  $\widetilde{\mathbf{I}}$  is not in the range  $[0, \text{nrows}(\widetilde{\mathbf{A}}))$ , the execution of `GrB_extract`  
 4449 ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING` mode,  
 4450 the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the result  
 4451 vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4452 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

4453 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .

4454 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$4455 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4456 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
4457 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$4458 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

$$4459 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

$$4460 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

4461 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4462 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
4463 using what is called a *standard vector mask and replace*. This is carried out under control of the  
4464 mask which acts as a “write mask”.

4465 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
4466 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$4467 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4470 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
4471 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
4472 mask are unchanged:

$$4473 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4474 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
4475 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
4476 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but  
4477 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
4478 sequence.

### 4479 4.3.7 assign: Modifying sub-graphs

4480 Assign the contents of a subset of a matrix or vector.

#### 4481 4.3.7.1 assign: Standard vector variant

4482 Assign values from one GraphBLAS vector to a subset of a vector as specified by a set of indices.  
4483 The size of the input vector is the same size as the index array provided.

## 4484 C Syntax

```
4485         GrB_Info GrB_assign(GrB_Vector      w,  
4486                             const GrB_Vector mask,  
4487                             const GrB_BinaryOp accum,  
4488                             const GrB_Vector u,  
4489                             const GrB_Index *indices,  
4490                             GrB_Index      nindices,  
4491                             const GrB_Descriptor desc);
```

## 4492 Parameters

4493        **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
4494        that may be accumulated with the result of the assign operation. On output, this  
4495        vector holds the results of the operation.

4496        **mask** (IN) An optional “write” mask that controls which results from this operation are  
4497        stored into the output vector **w**. The mask dimensions must match those of the  
4498        vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4499        of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
4500        in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4501        dimensions of **w**), **GrB\_NULL** should be specified.

4502        **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
4503        entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4504        specified.

4505        **u** (IN) The GraphBLAS vector whose contents are assigned to a subset of **w**.

4506        **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
4507        **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0  
4508        to **nindices** – 1, then **GrB\_ALL** should be specified. Regardless of execution mode  
4509        and return value, this array may be manipulated by the caller after this operation  
4510        returns without affecting any deferred computations for this operation. If this  
4511        array contains duplicate values, it implies in assignment of more than one value to  
4512        the same location which leads to undefined results.

4513        **nindices** (IN) The number of values in **indices** array. Must be equal to **size(u)**.

4514        **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
4515        should be specified. Non-default field/value pairs are listed as follows:  
4516

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.

## Return Values

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

**GrB\_PANIC** Unknown internal error.

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

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

**GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by a call to **new** (or **dup** for vector parameters).

**GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **indices** is greater than or equal to **size(w)**. In non-blocking mode, this can be reported as an execution error.

**GrB\_DIMENSION\_MISMATCH** mask and w dimensions are incompatible, or **nindices**  $\neq$  **size(u)**.

**GrB\_DOMAIN\_MISMATCH** The domains of the various vectors are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with **bool** (in the case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

**GrB\_NULL\_POINTER** Argument indices is a NULL pointer.

## Description

This variant of **GrB\_assign** computes the result of assigning elements from a source GraphBLAS vector to a destination GraphBLAS vector in a specific order:  $w(\text{indices}) = u$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $w(\text{indices}) = w(\text{indices}) \odot u$ . More explicitly:

$$\begin{aligned}
 w(\text{indices}[i]) &= u(i), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\
 w(\text{indices}[i]) &= w(\text{indices}[i]) \odot u(i), \forall i : 0 \leq i < \text{nindices}.
 \end{aligned}$$

4545 Logically, this operation occurs in three steps:

4546     **Setup** The internal vectors and mask used in the computation are formed and their domains  
4547             and dimensions are tested for compatibility.

4548     **Compute** The indicated computations are carried out.

4549     **Output** The result is written into the output vector, possibly under control of a mask.

4550 Up to three argument vectors are used in the `GrB_assign` operation:

- 4551     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 4552     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4553     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4554 The argument vectors and the accumulation operator (if provided) are tested for domain compati-  
4555 bility as follows:

- 4556     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
4557         must be from one of the pre-defined types of Table 3.2.
- 4558     2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .
- 4559     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
4560         of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accu-  
4561         mulation operator.

4562 Two domains are compatible with each other if values from one domain can be cast to values in  
4563 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4564 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4565 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
4566 error listed above is returned.

4567 From the arguments, the internal vectors, mask and index array used in the computation are formed  
4568 ( $\leftarrow$  denotes copy):

- 4569     1. Vector  $\widetilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 4570     2. One-dimensional mask,  $\widetilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 4571         (a) If `mask` = `GrB_NULL`, then  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 4572         (b) If `mask`  $\neq$  `GrB_NULL`,
    - 4573             i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 4574             ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
  - 4575         (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .

4576 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

4577 4. The internal index array,  $\tilde{\mathbf{I}}$ , is computed from argument indices as follows:

4578 (a) If `indices = GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$ .

4579 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$ .

4580 The internal vector and mask are checked for dimension compatibility. The following conditions  
4581 must hold:

4582 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

4583 2.  $\text{nindices} = \text{size}(\tilde{\mathbf{u}})$ .

4584 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
4585 match error listed above is returned.

4586 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4587 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4588 We are now ready to carry out the assign and any additional associated operations. We describe  
4589 this in terms of two intermediate vectors:

- 4590 •  $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{w}}$ .
- 4591 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4592 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$4593 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nindices} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4594 At this point, if any value of  $\tilde{\mathbf{I}}[i]$  is outside the valid range of indices for vector  $\tilde{\mathbf{w}}$ , computation  
4595 ends and the method returns the index-out-of-bounds error listed above. In `GrB_NONBLOCKING`  
4596 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
4597 result vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4598 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

- 4599 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$4600 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4601 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
4602 of  $\tilde{\mathbf{w}}$  ( $\text{ind}(\tilde{\mathbf{w}})$ ) and remove from it all the indices of  $\tilde{\mathbf{w}}$  that are in the set of indices being  
4603 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\text{ind}(\tilde{\mathbf{t}})$ ).

4604 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
4605 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$4606 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))),$$

$$4607 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4609 where the difference operator refers to set difference.

4610 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$4611 \quad \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4612 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
4613 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 4614 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 4615 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 4616 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

4618 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4620 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
4621 using what is called a *standard vector mask and replace*. This is carried out under control of the  
4622 mask which acts as a “write mask”.

4623 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
4624 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$4625 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4626 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
4627 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
4628 mask are unchanged:

$$4629 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4630 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
4631 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
4632 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but  
4633 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
4634 sequence.

#### 4635 4.3.7.2 assign: Standard matrix variant

4636 Assign values from one GraphBLAS matrix to a subset of a matrix as specified by a set of indices.  
4637 The dimensions of the input matrix are the same size as the row and column index arrays provided.

### 4638 C Syntax

```
4639      GrB_Info GrB_assign(GrB_Matrix      C,
4640                          const GrB_Matrix Mask,
4641                          const GrB_BinaryOp accum,
4642                          const GrB_Matrix A,
```

```

4643         const GrB_Index      *row_indices,
4644         GrB_Index             nrows,
4645         const GrB_Index      *col_indices,
4646         GrB_Index             ncols,
4647         const GrB_Descriptor  desc);

```

## 4648 Parameters

4649     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
4650     that may be accumulated with the result of the assign operation. On output, the  
4651     matrix holds the results of the operation.

4652     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
4653     stored into the output matrix **C**. The mask dimensions must match those of the  
4654     matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4655     of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
4656     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4657     dimensions of **C**), **GrB\_NULL** should be specified.

4658     **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
4659     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4660     specified.

4661     **A** (IN) The GraphBLAS matrix whose contents are assigned to a subset of **C**.

4662     **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**  
4663     that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** – 1,  
4664     then **GrB\_ALL** can be specified. Regardless of execution mode and return value,  
4665     this array may be manipulated by the caller after this operation returns without  
4666     affecting any deferred computations for this operation. If this array contains du-  
4667     plicate values, it implies assignment of more than one value to the same location  
4668     which leads to undefined results.

4669     **nrows** (IN) The number of values in the **row\_indices** array. Must be equal to **nrows(A)**  
4670     if **A** is not transposed, or equal to **ncols(A)** if **A** is transposed.

4671     **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns  
4672     of **C** that are assigned. If all columns of **C** are to be assigned in order from 0  
4673     to **ncols** – 1, then **GrB\_ALL** should be specified. Regardless of execution mode  
4674     and return value, this array may be manipulated by the caller after this operation  
4675     returns without affecting any deferred computations for this operation. If this  
4676     array contains duplicate values, it implies assignment of more than one value to  
4677     the same location which leads to undefined results.

4678     **ncols** (IN) The number of values in **col\_indices** array. Must be equal to **ncols(A)** if **A** is  
4679     not transposed, or equal to **nrows(A)** if **A** is transposed.



4680  
4681  
4682  
  
4683  
  
  
4684  
  
4685  
4686  
4687  
4688  
4689  
  
4690  
  
4691  
4692  
4693  
4694  
  
4695  
  
4696  
4697  
  
4698  
4699  
4700  
  
4701  
4702  
  
4703  
4704  
4705  
4706  
  
4707  
4708

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.

### 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(C), or a value in col\_indices is greater than or equal to ncols(C). In non-blocking mode, this can be reported as an execution error.

GrB\_DIMENSION\_MISMATCH Mask and C dimensions are incompatible, nrows  $\neq$  nrows(A), or ncols  $\neq$  ncols(A).

GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with bool (in the case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

GrB\_NULL\_POINTER Either argument row\_indices is a NULL pointer, argument col\_indices is a NULL pointer, or both.

## 4709 Description

4710 This variant of `GrB_assign` computes the result of assigning the contents of `A` to a subset of rows  
 4711 and columns in `C` in a specified order:  $C(\text{row\_indices}, \text{col\_indices}) = A$ ; or, if an optional binary  
 4712 accumulation operator ( $\odot$ ) is provided,  $C(\text{row\_indices}, \text{col\_indices}) = C(\text{row\_indices}, \text{col\_indices}) \odot$   
 4713 `A`. More explicitly (not accounting for an optional transpose of `A`):

$$\begin{aligned} & C(\text{row\_indices}[i], \text{col\_indices}[j]) = A(i, j), \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\ 4714 & C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot A(i, j), \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

4715 Logically, this operation occurs in three steps:

4716       Setup The internal matrices and mask used in the computation are formed and their domains  
 4717               and dimensions are tested for compatibility.

4718       Compute The indicated computations are carried out.

4719       Output The result is written into the output matrix, possibly under control of a mask.

4720 Up to three argument matrices are used in the `GrB_assign` operation:

- 4721 1.  $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4722 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)
- 4723 3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4724 The argument matrices and the accumulation operator (if provided) are tested for domain compat-  
 4725 ibility as follows:

- 4726 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 4727       must be from one of the pre-defined types of Table 3.2.
- 4728 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$ .
- 4729 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 4730       of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 4731       mulation operator.

4732 Two domains are compatible with each other if values from one domain can be cast to values in  
 4733 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 4734 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 4735 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
 4736 error listed above is returned.

4737 From the arguments, the internal matrices, mask, and index arrays used in the computation are  
 4738 formed ( $\leftarrow$  denotes copy):

- 4739 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 4740 2. Two-dimensional mask  $\tilde{\mathbf{M}}$  is computed from argument `Mask` as follows:
- 4741 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
4742  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
- 4743 (b) If `Mask  $\neq$  GrB_NULL`,
- 4744 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
4745  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
- 4746 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
4747  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
- 4748 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 4749 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 4750 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
- 4751 (a) If `row_indices = GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .
- 4752 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{row\_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .
- 4753 5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:
- 4754 (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$ .
- 4755 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \mathbf{col\_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$ .

4756 The internal matrices and mask are checked for dimension compatibility. The following conditions  
4757 must hold:

- 4758 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 4759 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 4760 3.  $\mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}$ .
- 4761 4.  $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}$ .

4762 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
4763 match error listed above is returned.

4764 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4765 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4766 We are now ready to carry out the assign and any additional associated operations. We describe  
4767 this in terms of two intermediate vectors:

- 4768 •  $\tilde{\mathbf{T}}$ : The matrix holding the contents from  $\tilde{\mathbf{A}}$  in their destination locations relative to  $\tilde{\mathbf{C}}$ .
- 4769 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

4770 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$4771 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \tilde{\mathbf{A}}(i, j)) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols} : (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle.$$

4772 At this point, if any value in the  $\tilde{\mathbf{I}}$  array is not in the range  $[0, \mathbf{nrows}(\tilde{\mathbf{C}}))$  or any value in the  
 4773  $\tilde{\mathbf{J}}$  array is not in the range  $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$ , the execution of `GrB_assign` ends and the index out-of-  
 4774 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred  
 4775 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from  
 4776 this point forward in the sequence.

4777 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows:

- 4778 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}}$  is defined as

$$4779 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 4780 \quad \{(i, j, Z_{ij}) \mid \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4781 The above expression defines the structure of matrix  $\tilde{\mathbf{Z}}$  as follows: We start with the structure  
 4782 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  
 4783 assigned ( $\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$ ). Finally, we add the structure of  $\tilde{\mathbf{T}}$  ( $\mathbf{ind}(\tilde{\mathbf{T}})$ ).

4784 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 4785 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$4786 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4787 \\ 4788 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}),$$

4789 where the difference operator refers to set difference.

- 4790 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$4791 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4792 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 4793 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$4794 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 4795 \\ 4796 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4797 \\ 4798 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4799 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4800 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 4801 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 4802 mask which acts as a “write mask”.

- If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in **C** on input to this operation are deleted and the content of the new output matrix, **C**, is defined as,

$$\mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are copied into the result matrix, **C**, and elements of **C** that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

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

### 4.3.7.3 assign: Column variant

Assign the contents a vector to a subset of elements in one column of a matrix. Note that since the output cannot be transposed, a different variant of **assign** is provided to assign to a row of a matrix.

## C Syntax

```
GrB_Info GrB_assign(GrB_Matrix      C,
                    const GrB_Vector mask,
                    const GrB_BinaryOp accum,
                    const GrB_Vector u,
                    const GrB_Index *row_indices,
                    GrB_Index        nrows,
                    GrB_Index        col_index,
                    const GrB_Descriptor desc);
```

## Parameters

**C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values that may be accumulated with the result of the assign operation. On output, this matrix holds the results of the operation.

**mask** (IN) An optional “write” mask that controls which results from this operation are stored into the specified column of the output matrix **C**. The mask dimensions must match those of a single column of the matrix **C**. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain of the **Mask** matrix must be of type

4836 bool or any of the predefined “built-in” types in Table 3.2. If the default mask  
 4837 is desired (i.e., a mask that is all true with the dimensions of a column of C),  
 4838 GrB\_NULL should be specified.

4839 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
 4840 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
 4841 specified.

4842 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a column  
 4843 of C.

4844 **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
 4845 the specified column of C that are to be assigned. If all elements of the column  
 4846 in C are to be assigned in order from index 0 to **nrows** – 1, then GrB\_ALL should  
 4847 be specified. Regardless of execution mode and return value, this array may be  
 4848 manipulated by the caller after this operation returns without affecting any de-  
 4849 ferred computations for this operation. If this array contains duplicate values, it  
 4850 implies in assignment of more than one value to the same location which leads to  
 4851 undefined results.

4852 **nrows** (IN) The number of values in **row\_indices** array. Must be equal to **size(u)**.

4853 **col\_index** (IN) The index of the column in C to assign. Must be in the range [0, **ncols(C)**).

4854 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 4855 should be specified. Non-default field/value pairs are listed as follows:

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> .

## 4858 Return Values

4859 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 4860 blocking mode, this indicates that the compatibility tests on  
 4861 dimensions and domains for the input arguments passed suc-  
 4862 cessfully. Either way, output matrix C is ready to be used in the  
 4863 next method of the sequence.

4864 **GrB\_PANIC** Unknown internal error.



4896 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4897 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain  
4898 compatibility as follows:

- 4899 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
4900 must be from one of the pre-defined types of Table 3.2.
- 4901 2.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .
- 4902 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
4903 of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
4904 mulation operator.

4905 Two domains are compatible with each other if values from one domain can be cast to values in  
4906 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4907 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4908 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
4909 error listed above is returned.

4910 The `col_index` parameter is checked for a valid value. The following condition must hold:

- 4911 1.  $0 \leq \text{col\_index} < \mathbf{ncols}(\mathbf{C})$

4912 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above  
4913 is returned.

4914 From the arguments, the internal vectors, `mask`, and index array used in the computation are  
4915 formed ( $\leftarrow$  denotes copy):

- 4916 1. The vector,  $\tilde{\mathbf{c}}$ , is extracted from a column of  $\mathbf{C}$  as follows:

$$4917 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \{(i, C_{ij}) \mid i : 0 \leq i < \mathbf{nrows}(\mathbf{C}), j = \text{col\_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

- 4918 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:

- 4919 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{nrows}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C})\} \rangle$ .
- 4920 (b) If `mask`  $\neq$  `GrB_NULL`,
  - 4921 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,
  - 4922 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
- 4923 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .

- 4924 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

- 4925 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:

- 4926 (a) If `row_indices` = `GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .



4927 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{row\_indices}[i]$ ,  $\forall i : 0 \leq i < \text{nrows}$ .

4928 The internal vectors, matrices, and masks are checked for dimension compatibility. The following  
4929 conditions must hold:

- 4930 1.  $\text{size}(\tilde{\mathbf{c}}) = \text{size}(\tilde{\mathbf{m}})$
- 4931 2.  $\text{nrows} = \text{size}(\tilde{\mathbf{u}})$ .

4932 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
4933 match error listed above is returned.

4934 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4935 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4936 We are now ready to carry out the assign and any additional associated operations. We describe  
4937 this in terms of two intermediate vectors:

- 4938 •  $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{c}}$ .
- 4939 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4940 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$4941 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nrows} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4942 At this point, if any value of  $\tilde{\mathbf{I}}[i]$  is outside the valid range of indices for vector  $\tilde{\mathbf{c}}$ , computation  
4943 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`  
4944 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
4945 result matrix,  $\mathbf{C}$ , is invalid from this point forward in the sequence.

4946 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

- 4947 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$4948 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4949 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
4950 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  
4951 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\text{ind}(\tilde{\mathbf{t}})$ ).

4952 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
4953 indices in  $\tilde{\mathbf{c}}$  and  $\tilde{\mathbf{t}}$ .

$$4954 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))),$$

$$4955 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4957 where the difference operator refers to set difference.

- If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$\langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$z_i = \tilde{\mathbf{c}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

Finally, the set of output values that make up the  $\tilde{\mathbf{z}}$  vector are written into the column of the final result matrix,  $\mathbf{C}(:, \text{col\_index})$ . This is carried out under control of the mask which acts as a “write mask”.

- If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}(:, \text{col\_index})$  on input to this operation are deleted and the new contents of the column is given by:

$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : j \neq \text{col\_index}\} \cup \{(i, \text{col\_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are copied into the column of the final result matrix,  $\mathbf{C}(:, \text{col\_index})$ , and elements of this column that fall outside the set indicated by the mask are unchanged:

$$\begin{aligned} \mathbf{L}(\mathbf{C}) = & \{(i, j, C_{ij}) : j \neq \text{col\_index}\} \cup \\ & \{(i, \text{col\_index}, \tilde{\mathbf{c}}(i)) : i \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \\ & \{(i, \text{col\_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}. \end{aligned}$$

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

#### 4.3.7.4 assign: Row variant

Assign the contents a vector to a subset of elements in one row of a matrix. Note that since the output cannot be transposed, a different variant of `assign` is provided to assign to a column of a matrix.

## 4988 C Syntax

```
4989         GrB_Info GrB_assign(GrB_Matrix      C,  
4990                             const GrB_Vector mask,  
4991                             const GrB_BinaryOp accum,  
4992                             const GrB_Vector u,  
4993                             GrB_Index      row_index,  
4994                             const GrB_Index *col_indices,  
4995                             GrB_Index      ncols,  
4996                             const GrB_Descriptor desc);
```

## 4997 Parameters

4998       **C** (INOUT) An existing GraphBLAS Matrix. On input, the matrix provides values  
4999       that may be accumulated with the result of the assign operation. On output, this  
5000       matrix holds the results of the operation.

5001       **mask** (IN) An optional “write” mask that controls which results from this operation are  
5002       stored into the specified row of the output matrix **C**. The mask dimensions must  
5003       match those of a single row of the matrix **C**. If the **GrB\_STRUCTURE** descriptor  
5004       is *not* set for the mask, the domain of the **Mask** matrix must be of type **bool** or  
5005       any of the predefined “built-in” types in Table 3.2. If the default mask is desired  
5006       (i.e., a mask that is all **true** with the dimensions of a row of **C**), **GrB\_NULL** should  
5007       be specified.

5008       **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
5009       entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5010       specified.

5011       **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a row of  
5012       **C**.

5013       **row\_index** (IN) The index of the row in **C** to assign. Must be in the range  $[0, \mathbf{nrows}(\mathbf{C})]$ .

5014       **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
5015       the specified row of **C** that are to be assigned. If all elements of the row in **C** are to  
5016       be assigned in order from index 0 to  $\mathbf{ncols} - 1$ , then **GrB\_ALL** should be specified.  
5017       Regardless of execution mode and return value, this array may be manipulated by  
5018       the caller after this operation returns without affecting any deferred computations  
5019       for this operation. If this array contains duplicate values, it implies in assignment  
5020       of more than one value to the same location which leads to undefined results.

5021       **ncols** (IN) The number of values in **col\_indices** array. Must be equal to **size(u)**.

5022       **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5023       should be specified. Non-default field/value pairs are listed as follows:  
5024

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output row in C is cleared (all elements removed) before result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of <b>mask</b> .

## Return Values

GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.
GrB_PANIC	Unknown internal error.
GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call <b>GrB_error()</b> to access any error messages generated by the implementation.
GrB_OUT_OF_MEMORY	Not enough memory available for operation.
GrB_UNINITIALIZED_OBJECT	One or more of the GraphBLAS objects has not been initialized by a call to <b>new</b> (or <b>dup</b> for vector or matrix parameters).
GrB_INVALID_INDEX	<b>row_index</b> is outside the allowable range (i.e., greater than <b>nrows(C)</b> ).
GrB_INDEX_OUT_OF_BOUNDS	A value in <b>col_indices</b> is greater than or equal to <b>ncols(C)</b> . In non-blocking mode, this can be reported as an execution error.
GrB_DIMENSION_MISMATCH	<b>mask</b> size and number of columns in C are not the same, or <b>ncols</b> $\neq$ <b>size(u)</b> .
GrB_DOMAIN_MISMATCH	The domains of the matrix and vector are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with <b>bool</b> (in the case where <b>desc[GrB_MASK].GrB_STRUCTURE</b> is not set).
GrB_NULL_POINTER	Argument <b>col_indices</b> is a NULL pointer.

## Description

This variant of **GrB\_assign** computes the result of assigning a subset of locations in a row of a GraphBLAS matrix (in a specific order) from the contents of a GraphBLAS vector:

5053  $C(\text{row\_index}, :) = u$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C(\text{row\_index}, :$   
 5054  $) = C(\text{row\_index}, :) \odot u$ . Taking order of `col_indices` into account it is more explicitly written as:

5055  $C(\text{row\_index}, \text{col\_indices}[j]) = u(j), \forall j : 0 \leq j < \text{ncols}, \text{ or}$   
 $C(\text{row\_index}, \text{col\_indices}[j]) = C(\text{row\_index}, \text{col\_indices}[j]) \odot u(j), \forall j : 0 \leq j < \text{ncols}$

5056 Logically, this operation occurs in three steps:

5057     **Setup** The internal matrices, vectors and mask used in the computation are formed and their  
 5058             domains and dimensions are tested for compatibility.

5059     **Compute** The indicated computations are carried out.

5060     **Output** The result is written into the output matrix, possibly under control of a mask.

5061 Up to three argument vectors and matrices are used in this `GrB_assign` operation:

- 5062     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5063     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 5064     3.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5065 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain  
 5066 compatibility as follows:

- 5067     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
 5068         must be from one of the pre-defined types of Table 3.2.
- 5069     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(u)$ .
- 5070     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 5071         of the accumulation operator and  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 5072         mulation operator.

5073 Two domains are compatible with each other if values from one domain can be cast to values in  
 5074 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 5075 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 5076 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
 5077 error listed above is returned.

5078 The `row_index` parameter is checked for a valid value. The following condition must hold:

- 5079     1.  $0 \leq \text{row\_index} < \mathbf{nrows}(C)$

5080 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above  
 5081 is returned.

5082 From the arguments, the internal vectors, mask, and index array used in the computation are  
 5083 formed ( $\leftarrow$  denotes copy):

5084 1. The vector,  $\tilde{\mathbf{c}}$ , is extracted from a row of  $\mathbf{C}$  as follows:

$$5085 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(j, C_{ij}) \mid \forall j : 0 \leq j < \mathbf{ncols}(\mathbf{C}), i = \text{row\_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

5086 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:

5087 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{ncols}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{ncols}(\mathbf{C})\} \rangle$ .

5088 (b) If `mask  $\neq$  GrB_NULL`,

5089 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,

5090 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .

5091 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .

5092 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

5093 4. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:

5094 (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$ .

5095 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \text{col\_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$ .

5096 The internal vectors, matrices, and masks are checked for dimension compatibility. The following  
5097 conditions must hold:

5098 1.  $\mathbf{size}(\tilde{\mathbf{c}}) = \mathbf{size}(\tilde{\mathbf{m}})$

5099 2.  $\mathbf{ncols} = \mathbf{size}(\tilde{\mathbf{u}})$ .

5100 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
5101 match error listed above is returned.

5102 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
5103 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5104 We are now ready to carry out the assign and any additional associated operations. We describe  
5105 this in terms of two intermediate vectors:

- 5106 •  $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{c}}$ .
- 5107 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

5108 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$5109 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{J}}[j], \tilde{\mathbf{u}}(j)) \mid \forall j, 0 \leq j < \mathbf{ncols} : j \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle.$$

5110 At this point, if any value of  $\tilde{\mathbf{J}}[j]$  is outside the valid range of indices for vector  $\tilde{\mathbf{c}}$ , computation  
5111 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`  
5112 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
5113 result matrix,  $\mathbf{C}$ , is invalid from this point forward in the sequence.

5114 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

5115 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$5116 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5117 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
 5118 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  
 5119 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\mathbf{ind}(\tilde{\mathbf{t}})$ ).

5120 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 5121 indices in  $\tilde{\mathbf{c}}$  and  $\tilde{\mathbf{t}}$ .

$$5122 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5123 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5124 where the difference operator refers to set difference.

5125 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$5126 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{c}}), \{(j, z_j) \mid j \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5127 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 5128 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$5130 \quad z_j = \tilde{\mathbf{c}}(j) \odot \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$5131 \quad z_j = \tilde{\mathbf{c}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5132 \quad z_j = \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

5133 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

5134 Finally, the set of output values that make up the  $\tilde{\mathbf{z}}$  vector are written into the column of the final  
 5135 result matrix,  $\mathbf{C}(\text{row\_index}, :)$ . This is carried out under control of the mask which acts as a “write  
 5136 mask”.

5137 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}(\text{row\_index}, :)$  on input to this  
 5138 operation are deleted and the new contents of the column is given by:

$$5139 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row\_index}\} \cup \{(\text{row\_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5140 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 5141 copied into the column of the final result matrix,  $\mathbf{C}(\text{row\_index}, :)$ , and elements of this column  
 5142 that fall outside the set indicated by the mask are unchanged:

$$5143 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row\_index}\} \cup$$

$$5144 \quad \{(\text{row\_index}, j, \tilde{\mathbf{c}}(j)) : j \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup$$

$$5145 \quad \{(\text{row\_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5146 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
 5147 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
 5148 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but may  
 5149 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 5152 4.3.7.5 assign: Constant vector variant[Scott: NEW CONTENT]

5153 Assign the same value to a specified subset of vector elements. With the use of GrB\_ALL, the entire  
5154 destination vector can be filled with the constant.

#### 5155 C Syntax

```
5156      GrB_Info GrB_assign(GrB_Vector      w,  
5157                          const GrB_Vector mask,  
5158                          const GrB_BinaryOp accum,  
5159                          <type>          val,  
5160                          const GrB_Index *indices,  
5161                          GrB_Index      nindices,  
5162                          const GrB_Descriptor desc);
```

```
5163      GrB_Info GrB_assign(GrB_Vector      w,  
5164                          const GrB_Vector mask,  
5165                          const GrB_BinaryOp accum,  
5166                          const GrB_Scalar s,  
5167                          const GrB_Index *indices,  
5168                          GrB_Index      nindices,  
5169                          const GrB_Descriptor desc);
```

#### 5170 Parameters

5171 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
5172 that may be accumulated with the result of the assign operation. On output, this  
5173 vector holds the results of the operation.

5174 **mask** (IN) An optional “write” mask that controls which results from this operation are  
5175 stored into the output vector **w**. The mask dimensions must match those of the  
5176 vector **w**. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
5177 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
5178 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5179 dimensions of **w**), GrB\_NULL should be specified.

5180 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
5181 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
5182 specified.

5183 **val** (IN) Scalar value to assign to (a subset of) **w**.

5184 **s** (IN) Scalar value to assign to (a subset of) **w**.

5185 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
5186 **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0



5187 to `nindices - 1`, then `GrB_ALL` should be specified. Regardless of execution mode  
5188 and return value, this array may be manipulated by the caller after this operation  
5189 returns without affecting any deferred computations for this operation. In this  
5190 variant, the specific order of the values in the array has no effect on the result.  
5191 Unlike other variants, if there are duplicated values in this array the result is still  
5192 defined.

5193 **nindices** (IN) The number of values in `indices` array. Must be in the range: `[0, size(w)]`. If  
5194 `nindices` is zero, the operation becomes a NO-OP.

5195 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
5196 should be specified. Non-default field/value pairs are listed as follows:

5197

Param	Field	Value	Description
<code>w</code>	<code>GrB_OUTP</code>	<code>GrB_REPLACE</code>	Output vector <code>w</code> is cleared (all elements removed) before the result is stored in it.
<code>mask</code>	<code>GrB_MASK</code>	<code>GrB_STRUCTURE</code>	The write mask is constructed from the structure (pattern of stored values) of the input <code>mask</code> vector. The stored values are not examined.
<code>mask</code>	<code>GrB_MASK</code>	<code>GrB_COMP</code>	Use the complement of <code>mask</code> .

5198

## 5199 Return Values

5200 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
5201 blocking mode, this indicates that the compatibility tests on  
5202 dimensions and domains for the input arguments passed suc-  
5203 cessfully. Either way, output vector `w` is ready to be used in the  
5204 next method of the sequence.

5205 **GrB\_PANIC** Unknown internal error.

5206 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
5207 opaque GraphBLAS objects (input or output) is in an invalid  
5208 state caused by a previous execution error. Call `GrB_error()` to  
5209 access any error messages generated by the implementation.

5210 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

5211 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized  
5212 by a call to `new` (or `dup` for vector parameters).

5213 **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in `indices` is greater than or equal to `size(w)`. In non-  
5214 blocking mode, this can be reported as an execution error.

5215 **GrB\_DIMENSION\_MISMATCH** `mask` and `w` dimensions are incompatible, or `nindices` is not less  
5216 than `size(w)`.



5247 4. If **accum** is not **GrB\_NULL**, then either **D(val)** or **D(s)**, depending on the signature of the  
 5248 method, must be compatible with **D<sub>in2</sub>(accum)** of the accumulation operator.

5249 Two domains are compatible with each other if values from one domain can be cast to values in  
 5250 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 5251 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 5252 any compatibility rule above is violated, execution of **GrB\_assign** ends and the domain mismatch  
 5253 error listed above is returned.

5254 From the arguments, the internal vectors, mask and index array used in the computation are formed  
 5255 ( $\leftarrow$  denotes copy):

- 5256 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5257 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument **mask** as follows:
  - 5258 (a) If **mask** = **GrB\_NULL**, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 5259 (b) If **mask**  $\neq$  **GrB\_NULL**,
    - 5260 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 5261 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}(\mathbf{mask})(i) = \mathbf{true})\} \rangle$ .
  - 5262 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 5263 3. Scalar  $\tilde{s} \leftarrow s$  (**GrB\_Scalar** version only).
- 5264 4. The internal index array,  $\tilde{\mathbf{I}}$ , is computed from argument **indices** as follows:
  - 5265 (a) If **indices** = **GrB\_ALL**, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nindices}$ .
  - 5266 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{indices}[i], \forall i : 0 \leq i < \mathbf{nindices}$ .

5267 The internal vector and mask are checked for dimension compatibility. The following conditions  
 5268 must hold:

- 5269 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5270 2.  $0 \leq \mathbf{nindices} \leq \mathbf{size}(\tilde{\mathbf{w}})$ .

5271 If any compatibility rule above is violated, execution of **GrB\_assign** ends and the dimension mis-  
 5272 match error listed above is returned.

5273 From this point forward, in **GrB\_NONBLOCKING** mode, the method can optionally exit with  
 5274 **GrB\_SUCCESS** return code and defer any computation and/or execution error codes.

5275 We are now ready to carry out the assign and any additional associated operations. We describe  
 5276 this in terms of two intermediate vectors:

- 5277 •  $\tilde{\mathbf{t}}$ : The vector holding the copies of the scalar, either **val** or  $\tilde{s}$ , in their destination locations  
 5278 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. If a non-opaque scalar  $\mathbf{val}$  is provided:

$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{val}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

Correspondingly, if a non-empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 1$ ):

$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}(\tilde{s})) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

Finally, if an empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 0$ ):

$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \emptyset \rangle.$$

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 is not in the range  $[0, \mathbf{size}(\tilde{\mathbf{w}}))$ , the execution of `GrB_assign` ends and the index out-of-bounds error listed above is generated. 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. We note that in this case of assigning a constant,  $\{\tilde{\mathbf{I}}[k], \forall k\}$  and  $\mathbf{ind}(\tilde{\mathbf{t}})$  are identical.

- If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$\langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

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  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

5314 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 5315 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 5316 mask which acts as a “write mask”.

- 5317 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are  
 5318 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$5319 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 5320 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 5321 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 5322 mask are unchanged:

$$5323 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5324 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 5325 of vector  $\mathbf{w}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 5326 exits with return value GrB\_SUCCESS and the new content of vector  $\mathbf{w}$  is as defined above but  
 5327 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 5328 sequence.

#### 5329 4.3.7.6 assign: Constant matrix variant[Scott: NEW CONTENT]

5330 Assign the same value to a specified subset of matrix elements. With the use of GrB\_ALL, the  
 5331 entire destination matrix can be filled with the constant.

### 5332 C Syntax

```
5333      GrB_Info GrB_assign(GrB_Matrix      C,
5334                        const GrB_Matrix  Mask,
5335                        const GrB_BinaryOp accum,
5336                        <type>            val,
5337                        const GrB_Index    *row_indices,
5338                        GrB_Index          nrows,
5339                        const GrB_Index    *col_indices,
5340                        GrB_Index          ncols,
5341                        const GrB_Descriptor desc);
```

```
5342      GrB_Info GrB_assign(GrB_Matrix      C,
5343                        const GrB_Matrix  Mask,
5344                        const GrB_BinaryOp accum,
5345                        const GrB_Scalar   s,
5346                        const GrB_Index    *row_indices,
5347                        GrB_Index          nrows,
```

```

5348         const GrB_Index      *col_indices,
5349         GrB_Index             ncols,
5350         const GrB_Descriptor   desc);

```

## 5351 Parameters

5352     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
5353     that may be accumulated with the result of the assign operation. On output, the  
5354     matrix holds the results of the operation.

5355     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
5356     stored into the output matrix **C**. The mask dimensions must match those of the  
5357     matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5358     of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
5359     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5360     dimensions of **C**), **GrB\_NULL** should be specified.

5361     **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
5362     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5363     specified.

5364     **val** (IN) Scalar value to assign to (a subset of) **C**.

5365     **s** (IN) Scalar value to assign to (a subset of) **C**.

5366     **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**  
5367     that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** − 1,  
5368     then **GrB\_ALL** can be specified. Regardless of execution mode and return value,  
5369     this array may be manipulated by the caller after this operation returns without  
5370     affecting any deferred computations for this operation. Unlike other variants, if  
5371     there are duplicated values in this array the result is still defined.

5372     **nrows** (IN) The number of values in **row\_indices** array. Must be in the range: [0, **nrows**(**C**)].  
5373     If **nrows** is zero, the operation becomes a NO-OP.

5374     **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns of **C**  
5375     that are assigned. If all columns of **C** are to be assigned in order from 0 to **ncols** − 1,  
5376     then **GrB\_ALL** should be specified. Regardless of execution mode and return value,  
5377     this array may be manipulated by the caller after this operation returns without  
5378     affecting any deferred computations for this operation. Unlike other variants, if  
5379     there are duplicated values in this array the result is still defined.

5380     **ncols** (IN) The number of values in **col\_indices** array. Must be in the range: [0, **ncols**(**C**)].  
5381     If **ncols** is zero, the operation becomes a NO-OP.

5382     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5383     should be specified. Non-default field/value pairs are listed as follows:  
5384

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of Mask.

## Return Values

GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.
GrB_PANIC	Unknown internal error.
GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call <code>GrB_error()</code> to access any error messages generated by the implementation.
GrB_OUT_OF_MEMORY	Not enough memory available for the operation.
GrB_UNINITIALIZED_OBJECT	One or more of the GraphBLAS objects has not been initialized by a call to <code>new</code> (or <code>dup</code> for vector parameters).
GrB_INDEX_OUT_OF_BOUNDS	A value in <code>row_indices</code> is greater than or equal to <code>nrows(C)</code> , or a value in <code>col_indices</code> is greater than or equal to <code>ncols(C)</code> . In non-blocking mode, this can be reported as an execution error.
GrB_DIMENSION_MISMATCH	Mask and C dimensions are incompatible, <code>nrows</code> is not less than <code>nrows(C)</code> , or <code>ncols</code> is not less than <code>ncols(C)</code> .
GrB_DOMAIN_MISMATCH	The domains of the matrix and scalar are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with <code>bool</code> (in the case where <code>desc[GrB_MASK].GrB_STRUCTURE</code> is not set).
GrB_NULL_POINTER	Either argument <code>row_indices</code> is a NULL pointer, argument <code>col_indices</code> is a NULL pointer, or both.

## Description

This variant of `GrB_assign` computes the result of assigning a constant scalar value – either `val` or `s` – to locations in a destination GraphBLAS matrix: Either `C(row_indices, col_indices) = val`

5414 or  $C(\text{row\_indices}, \text{col\_indices}) = s$  is performed. If an optional binary accumulation operator  
 5415  $(\odot)$  is provided, then either  $C(\text{row\_indices}, \text{col\_indices}) = C(\text{row\_indices}, \text{col\_indices}) \odot \text{val}$  or  
 5416  $C(\text{row\_indices}, \text{col\_indices}) = C(\text{row\_indices}, \text{col\_indices}) \odot s$  is performed. More explicitly, if a  
 5417 non-opaque value  $\text{val}$  is provided:

$$\begin{aligned} & C(\text{row\_indices}[i], \text{col\_indices}[j]) = \text{val}, \text{ or} \\ 5418 \quad & C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot \text{val} \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5419 Correspondingly, if a `GrB_Scalar`  $s$  is provided:

$$\begin{aligned} & C(\text{row\_indices}[i], \text{col\_indices}[j]) = s, \text{ or} \\ 5420 \quad & C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot s \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5421 Logically, this operation occurs in three steps:

5422     Setup The internal vectors and mask used in the computation are formed and their domains  
 5423     and dimensions are tested for compatibility.

5424     Compute The indicated computations are carried out.

5425     Output The result is written into the output matrix, possibly under control of a mask.

5426 Up to two argument matrices are used in the `GrB_assign` operation:

- 5427     1.  $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5428     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)

5429 The argument scalar, matrices, and the accumulation operator (if provided) are tested for domain  
 5430 compatibility as follows:

- 5431     1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 5432     must be from one of the pre-defined types of Table 3.2.
- 5433     2.  $\mathbf{D}(C)$  must be compatible with either  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(s)$ , depending on the signature of the  
 5434     method.
- 5435     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 5436     of the accumulation operator.
- 5437     4. If `accum` is not `GrB_NULL`, then either  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(s)$ , depending on the signature of the  
 5438     method, must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.



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

From the arguments, the internal matrices, index arrays, and mask used in the computation are formed ( $\leftarrow$  denotes copy):

1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
2. Two-dimensional mask  $\tilde{\mathbf{M}}$  is computed from argument `Mask` as follows:
  - (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - (b) If `Mask  $\neq$  GrB_NULL`,
    - i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
3. Scalar  $\tilde{s} \leftarrow s$  (`GrB_Scalar` version only).
4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
  - (a) If `row_indices = GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .
  - (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{row\_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .
5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:
  - (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$ .
  - (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \mathbf{col\_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$ .

The internal matrix and mask are checked for dimension compatibility. The following conditions must hold:

1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
3.  $0 \leq \mathbf{nrows} \leq \mathbf{nrows}(\tilde{\mathbf{C}})$ .
4.  $0 \leq \mathbf{ncols} \leq \mathbf{ncols}(\tilde{\mathbf{C}})$ .

If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mismatch error listed above is returned.

5471 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 5472 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5473 We are now ready to carry out the assign and any additional associated operations. We describe  
 5474 this in terms of two intermediate matrices:

- 5475 •  $\tilde{\mathbf{T}}$ : The matrix holding the copies of the scalar, either `val` or  $\tilde{s}$ , in their destination locations  
 5476 relative to  $\tilde{\mathbf{C}}$ .
- 5477 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

5478 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows. If a non-opaque scalar `val` is provided:

$$\begin{aligned} \tilde{\mathbf{T}} = & \langle \mathbf{D}(\text{val}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 5479 & \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle. \end{aligned}$$

5480 Correspondingly, if a non-empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e., `size`( $\tilde{s}$ ) = 1):

$$\begin{aligned} \tilde{\mathbf{T}} = & \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 5481 & \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}(\tilde{s})) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle. \end{aligned}$$

5482 Finally, if an empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e., `size`( $\tilde{s}$ ) = 0):

$$5483 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \emptyset \rangle.$$

5484 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  
 5485 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  
 5486  $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$ , the execution of `GrB_assign` ends and the index out-of-bounds error listed above is  
 5487 generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating  
 5488 `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from this point forward in the  
 5489 sequence.

5490 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows:

- 5491 • If `accum` = `GrB_NULL`, then  $\tilde{\mathbf{Z}}$  is defined as

$$\begin{aligned} \tilde{\mathbf{Z}} = & \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 5492 & \{(i, j, Z_{ij}) \mid \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle. \end{aligned}$$

5494 The above expression defines the structure of matrix  $\tilde{\mathbf{Z}}$  as follows: We start with the structure  
 5495 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  
 5496 assigned ( $\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$ ). Finally, we add the structure of  $\tilde{\mathbf{T}}$  ( $\mathbf{ind}(\tilde{\mathbf{T}})$ ).

5497 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 5498 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 5499 \quad Z_{ij} = & \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5500 & \\ 5501 \quad & \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}), \end{aligned}$$

5502 where the difference operator refers to set difference. We note that, in this particular case of  
 5503 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.

5504 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

5505 
$$\langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5506 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
5507 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

5508 
$$Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

5509

5510 
$$Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5511

5512 
$$Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5513 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

5514 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
5515 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
5516 mask which acts as a “write mask”.

5517 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
5518 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

5519 
$$\mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5520 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
5521 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
5522 mask are unchanged:

5523 
$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5524 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
5525 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
5526 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
5527 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
5528 sequence.

#### 5529 4.3.8 apply: Apply a function to the elements of an object

5530 Computes the transformation of the values of the elements of a vector or a matrix using a unary  
5531 function, or a binary function where one argument is bound to a scalar.

##### 5532 4.3.8.1 apply: Vector variant

5533 Computes the transformation of the values of the elements of a vector using a unary function.

## 5534 C Syntax

```

5535         GrB_Info GrB_apply(GrB_Vector          w,
5536                           const GrB_Vector      mask,
5537                           const GrB_BinaryOp     accum,
5538                           const GrB_UnaryOp      op,
5539                           const GrB_Vector      u,
5540                           const GrB_Descriptor   desc);

```

## 5541 Parameters

5542     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
5543     that may be accumulated with the result of the apply operation. On output, this  
5544     vector holds the results of the operation.

5545     **mask** (IN) An optional “write” mask that controls which results from this operation are  
5546     stored into the output vector **w**. The mask dimensions must match those of the  
5547     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5548     of the mask vector must be of type **bool** or any of the predefined “built-in” types  
5549     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5550     dimensions of **w**), **GrB\_NULL** should be specified.

5551     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
5552     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5553     specified.

5554     **op** (IN) A unary operator applied to each element of input vector **u**.

5555     **u** (IN) The GraphBLAS vector to which the unary function is applied.

5556     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5557     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> .

## 5560 Return Values

5561     **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
5562     blocking mode, this indicates that the compatibility tests on di-  
5563     mensions and domains for the input arguments passed successfully.

5564 Either way, output vector  $w$  is ready to be used in the next method  
 5565 of the sequence.

5566 **GrB\_PANIC** Unknown internal error.

5567 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 5568 GraphBLAS objects (input or output) is in an invalid state caused  
 5569 by a previous execution error. Call **GrB\_error()** to access any error  
 5570 messages generated by the implementation.

5571 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

5572 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 5573 a call to **new** (or **dup** for vector parameters).

5574 **GrB\_DIMENSION\_MISMATCH**  $mask$ ,  $w$  and/or  $u$  dimensions are incompatible.

5575 **GrB\_DOMAIN\_MISMATCH** The domains of the various vectors are incompatible with the corre-  
 5576 sponding domains of the accumulation operator or unary function,  
 5577 or the mask's domain is not compatible with **bool** (in the case where  
 5578  $desc[GrB\_MASK].GrB\_STRUCTURE$  is not set).

## 5579 Description

5580 This variant of **GrB\_apply** computes the result of applying a unary function to the elements of a  
 5581 GraphBLAS vector:  $w = f(u)$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  
 5582  $w = w \odot f(u)$ .

5583 Logically, this operation occurs in three steps:

5584 **Setup** The internal vectors and mask used in the computation are formed and their domains  
 5585 and dimensions are tested for compatibility.

5586 **Compute** The indicated computations are carried out.

5587 **Output** The result is written into the output vector, possibly under control of a mask.

5588 Up to three argument vectors are used in this **GrB\_apply** operation:

- 5589 1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 5590 2.  $mask = \langle \mathbf{D}(mask), \mathbf{size}(mask), \mathbf{L}(mask) = \{(i, m_i)\} \rangle$  (optional)
- 5591 3.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5592 The argument vectors, unary operator and the accumulation operator (if provided) are tested for  
 5593 domain compatibility as follows:

- 5594 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
5595 must be from one of the pre-defined types of Table 3.2.
- 5596 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the unary operator.
- 5597 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
5598 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the unary operator must be compatible with  
5599  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 5600 4.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in}(\text{op})$ .

5601 Two domains are compatible with each other if values from one domain can be cast to values in  
5602 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
5603 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
5604 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
5605 error listed above is returned.

5606 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
5607 denotes copy):

- 5608 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5609 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 5610 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .
  - 5611 (b) If `mask`  $\neq$  `GrB_NULL`,
    - 5612 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
    - 5613 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 5614 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 5615 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

5616 The internal vectors and masks are checked for dimension compatibility. The following conditions  
5617 must hold:

- 5618 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$
- 5619 2.  $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$ .

5620 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
5621 error listed above is returned.

5622 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
5623 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5624 We are now ready to carry out the apply and any additional associated operations. We describe  
5625 this in terms of two intermediate vectors:

- $\tilde{\mathbf{t}}$ : The vector holding the result from applying the unary operator to the input vector  $\tilde{\mathbf{u}}$ .
- $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f(\tilde{\mathbf{u}}(i))) \mid \forall i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle,$$

where  $f = \mathbf{f}(\text{op})$ .

The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ , using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a “write mask”.

- If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

- If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

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

#### 4.3.8.2 apply: Matrix variant

Computes the transformation of the values of the elements of a matrix using a unary function.

#### C Syntax

```
GrB_Info GrB_apply(GrB_Matrix      C,
                  const GrB_Matrix  Mask,
                  const GrB_BinaryOp accum,
                  const GrB_UnaryOp  op,
                  const GrB_Matrix  A,
                  const GrB_Descriptor desc);
```

#### Parameters

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

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

**accum** (IN) An optional binary operator used for accumulating entries into existing C entries. If assignment rather than accumulation is desired, `GrB_NULL` should be specified.

**op** (IN) A unary operator applied to each element of input matrix A.

**A** (IN) The GraphBLAS matrix to which the unary function is applied.

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

Param	Field	Value	Description
C	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 <b>Mask</b> matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of <b>Mask</b> .
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.



## 5686 Return Values

5687	<b>GrB_SUCCESS</b>	In blocking mode, the operation completed successfully. In non-
5688		blocking mode, this indicates that the compatibility tests on
5689		dimensions and domains for the input arguments passed suc-
5690		cessfully. Either way, output matrix C is ready to be used in the
5691		next method of the sequence.
5692	<b>GrB_PANIC</b>	Unknown internal error.
5693	<b>GrB_INVALID_OBJECT</b>	This is returned in any execution mode whenever one of the
5694		opaque GraphBLAS objects (input or output) is in an invalid
5695		state caused by a previous execution error. Call <b>GrB_error()</b> to
5696		access any error messages generated by the implementation.
5697	<b>GrB_OUT_OF_MEMORY</b>	Not enough memory available for the operation.
5698	<b>GrB_UNINITIALIZED_OBJECT</b>	One or more of the GraphBLAS objects has not been initialized
5699		by a call to <b>new</b> (or <b>Matrix_dup</b> for matrix parameters).
5700	<b>GrB_DIMENSION_MISMATCH</b>	Mask and C dimensions are incompatible, $\mathbf{nrows} \neq \mathbf{nrows}(C)$ , or
5701		$\mathbf{ncols} \neq \mathbf{ncols}(C)$ .
5702	<b>GrB_DOMAIN_MISMATCH</b>	The domains of the various matrices are incompatible with the
5703		corresponding domains of the accumulation operator or unary
5704		function, or the mask's domain is not compatible with <b>bool</b> (in
5705		the case where <b>desc[GrB_MASK].GrB_STRUCTURE</b> is not set).

## 5706 Description

5707 This variant of **GrB\_apply** computes the result of applying a unary function to the elements of a  
 5708 GraphBLAS matrix:  $C = f(A)$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  
 5709  $C = C \odot f(A)$ .

5710 Logically, this operation occurs in three steps:

5711 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 5712 and dimensions are tested for compatibility.

5713 **Compute** The indicated computations are carried out.

5714 **Output** The result is written into the output matrix, possibly under control of a mask.

5715 Up to three argument matrices are used in the **GrB\_apply** operation:

- 5716 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5717 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)

5718 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

5719 The argument matrices, unary operator and the accumulation operator (if provided) are tested for  
5720 domain compatibility as follows:

- 5721 1. If **Mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\text{Mask})$   
5722 must be from one of the pre-defined types of Table 3.2.
- 5723 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the unary operator.
- 5724 3. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
5725 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the unary operator must be compatible with  
5726  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 5727 4.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in}(\text{op})$  of the unary operator.

5728 Two domains are compatible with each other if values from one domain can be cast to values in  
5729 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
5730 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
5731 any compatibility rule above is violated, execution of **GrB\_apply** ends and the domain mismatch  
5732 error listed above is returned.

5733 From the argument matrices, the internal matrices, mask, and index arrays used in the computation  
5734 are formed ( $\leftarrow$  denotes copy):

- 5735 1. Matrix  $\tilde{C} \leftarrow C$ .
- 5736 2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument **Mask** as follows:
  - 5737 (a) If **Mask** = **GrB\_NULL**, then  $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$   
5738  $j < \mathbf{ncols}(C)\} \rangle$ .
  - 5739 (b) If **Mask**  $\neq$  **GrB\_NULL**,
    - 5740 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
5741  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
    - 5742 ii. Otherwise,  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
5743  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
  - 5744 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\tilde{M} \leftarrow \neg \tilde{M}$ .
- 5745 3. Matrix  $\tilde{A} \leftarrow \text{desc[GrB_INP0].GrB_TRAN} ? A^T : A$ .

5746 The internal matrices and mask are checked for dimension compatibility. The following conditions  
5747 must hold:

- 5748 1.  $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{M})$ .
- 5749 2.  $\mathbf{ncols}(\tilde{C}) = \mathbf{ncols}(\tilde{M})$ .
- 5750 3.  $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{A})$ .

5751 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

5752 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
5753 error listed above is returned.

5754 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
5755 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5756 We are now ready to carry out the apply and any additional associated operations. We describe  
5757 this in terms of two intermediate matrices:

- 5758 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the unary operator to the input matrix  $\tilde{\mathbf{A}}$ .
- 5759 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

5760 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$5761 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j))) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

5762 where  $f = \mathbf{f}(\mathbf{op})$ .

5763 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 5764 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 5765 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$5766 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5767 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
5768 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$5769 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$5770 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$5771 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5772 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

5773 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
5774 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
5775 mask which acts as a “write mask”.

- 5776 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
5777 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$5780 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 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:

$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\text{ind}(\mathbf{C}) \cap \text{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\text{ind}(\tilde{\mathbf{Z}}) \cap \text{ind}(\tilde{\mathbf{M}}))\}.$$

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.

#### 4.3.8.3 apply: Vector-BinaryOp variants[Scott: NEW CONTENT]

Computes the transformation of the values of the stored elements of a vector 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 vector are passed as the second argument. In the *bind-second* variant, the elements of the vector are passed as the first argument and the specified scalar value is passed as the second argument. The scalar can be passed either as a non-opaque variable or as a GrB\_Scalar object.

#### C Syntax

```
// bind-first + scalar value
GrB_Info GrB_apply(GrB_Vector          w,
                   const GrB_Vector     mask,
                   const GrB_BinaryOp   accum,
                   const GrB_BinaryOp   op,
                   <type>               val,
                   const GrB_Vector     u,
                   const GrB_Descriptor desc);
```

```
// bind-first + GraphBLAS scalar
GrB_Info GrB_apply(GrB_Vector          w,
                   const GrB_Vector     mask,
                   const GrB_BinaryOp   accum,
                   const GrB_BinaryOp   op,
                   const GrB_Scalar     s,
                   const GrB_Vector     u,
                   const GrB_Descriptor desc);
```

```
// bind-second + scalar value
GrB_Info GrB_apply(GrB_Vector          w,
                   const GrB_Vector     mask,
```

```

5817         const GrB_BinaryOp      accum,
5818         const GrB_BinaryOp      op,
5819         const GrB_Vector        u,
5820         <type>                  val,
5821         const GrB_Descriptor    desc);

5822 // bind-second + GraphBLAS scalar
5823 GrB_Info GrB_apply(GrB_Vector      w,
5824                   const GrB_Vector mask,
5825                   const GrB_BinaryOp accum,
5826                   const GrB_BinaryOp op,
5827                   const GrB_Vector u,
5828                   const GrB_Scalar s,
5829                   const GrB_Descriptor desc);

```

## 5830 Parameters

5831     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
5832     that may be accumulated with the result of the apply operation. On output, this  
5833     vector holds the results of the operation.

5834     **mask** (IN) An optional “write” mask that controls which results from this operation are  
5835     stored into the output vector **w**. The mask dimensions must match those of the  
5836     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5837     of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
5838     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5839     dimensions of **w**), **GrB\_NULL** should be specified.

5840     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
5841     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5842     specified.

5843     **op** (IN) A binary operator applied to each element of input vector, **u**, and the scalar  
5844     value, **val**.

5845     **u** (IN) The GraphBLAS vector whose elements are passed to the binary operator as  
5846     the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)  
5847     argument in the *bind-second* variant.

5848     **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)  
5849     argument in the *bind-first* variant, or the right-hand (second) argument in the  
5850     *bind-second* variant.

5851     **s** (IN) A GraphBLAS scalar that is passed to the binary operator as the left-hand  
5852     (first) argument in the *bind-first* variant, or the right-hand (second) argument in  
5853     the *bind-second* variant. It must not be empty.

5854 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
5855 should be specified. Non-default field/value pairs are listed as follows:

5856

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector <b>w</b> is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of <b>mask</b> .

5857

## 5858 Return Values

5859 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
5860 blocking mode, this indicates that the compatibility tests on di-  
5861 mensions and domains for the input arguments passed successfully.  
5862 Either way, output vector **w** is ready to be used in the next method  
5863 of the sequence.

5864 GrB\_PANIC Unknown internal error.

5865 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
5866 GraphBLAS objects (input or output) is in an invalid state caused  
5867 by a previous execution error. Call GrB\_error() to access any error  
5868 messages generated by the implementation.

5869 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

5870 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
5871 a call to new (or dup for vector parameters).

5872 GrB\_DIMENSION\_MISMATCH mask, w and/or u dimensions are incompatible.

5873 GrB\_DOMAIN\_MISMATCH The domains of the various vectors and scalar are incompatible with  
5874 the corresponding domains of the binary operator or accumulation  
5875 operator, or the mask's domain is not compatible with bool (in the  
5876 case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

5877 GrB\_EMPTY\_OBJECT The GrB\_Scalar **s** used in the call is empty (**nvals(s) = 0**) and  
5878 therefore a value cannot be passed to the binary operator.

## 5879 Description

5880 This variant of GrB\_apply computes the result of applying a binary operator to the elements of a  
5881 GraphBLAS vector each composed with a scalar constant, either **val** or **s**:

5882                   bind-first:      $w = f(\text{val}, u)$  or  $w = f(s, u)$

5883                   bind-second:     $w = f(u, \text{val})$  or  $w = f(u, s)$ ,

5884 or if an optional binary accumulation operator ( $\odot$ ) is provided:

5885                   bind-first:      $w = w \odot f(\text{val}, u)$  or  $w = w \odot f(s, u)$

5886                   bind-second:     $w = w \odot f(u, \text{val})$  or  $w = w \odot f(u, s)$ .

5887 Logically, this operation occurs in three steps:

5888     **Setup** The internal vectors and mask used in the computation are formed and their domains  
5889             and dimensions are tested for compatibility.

5890     **Compute** The indicated computations are carried out.

5891     **Output** The result is written into the output vector, possibly under control of a mask.

5892 Up to three argument vectors are used in this GrB\_apply operation:

5893     1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$

5894     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)

5895     3.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5896 The argument scalar, vectors, binary operator and the accumulation operator (if provided) are  
5897 tested for domain compatibility as follows:

5898     1. If **mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{mask})$   
5899         must be from one of the pre-defined types of Table 3.2.

5900     2.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the binary operator.

5901     3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
5902         of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the binary operator must be compatible with  
5903          $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.

5904     4.  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.

5905     5. If bind-first:

5906         (a)  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the binary operator.

5907         (b) If the non-opaque scalar **val** is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$   
5908             of the binary operator.

5909         (c) If the GrB\_Scalar **s** is provided, then  $\mathbf{D}(s)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the  
5910             binary operator.

- 5911 6. If bind-second:
- 5912 (a)  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the binary operator.
- 5913 (b) If the non-opaque scalar  $\mathbf{val}$  is provided, then  $\mathbf{D}(\mathbf{val})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$
- 5914 of the binary operator.
- 5915 (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
- 5916 binary operator.

5917 Two domains are compatible with each other if values from one domain can be cast to values in

5918 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all

5919 compatible with each other. A domain from a user-defined type is only compatible with itself. If

5920 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch

5921 error listed above is returned.

5922 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$

5923 denotes copy):

- 5924 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5925 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
- 5926 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
- 5927 (b) If `mask  $\neq$  GrB_NULL`,
- 5928 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
- 5929 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
- 5930 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 5931 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 5932 4. Scalar  $\tilde{\mathbf{s}} \leftarrow \mathbf{s}$  (GraphBLAS scalar case).

5933 The internal vectors and masks are checked for dimension compatibility. The following conditions

5934 must hold:

- 5935 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5936 2.  $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{w}})$ .

5937 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch

5938 error listed above is returned.

5939 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with

5940 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5941 If an empty `GrB_Scalar`  $\tilde{\mathbf{s}}$  is provided ( $\mathbf{nvals}(\tilde{\mathbf{s}}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.

5942 If a non-empty `GrB_Scalar`,  $\tilde{\mathbf{s}}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{\mathbf{s}}) = 1$ ), we then create an internal variable

5943 `val` with the same domain as  $\tilde{\mathbf{s}}$  and set `val = val( $\tilde{\mathbf{s}}$ )`.

5944 We are now ready to carry out the apply and any additional associated operations. We describe

5945 this in terms of two intermediate vectors:



- $\tilde{\mathbf{t}}$ : The vector holding the result from applying the binary operator to the input vector  $\tilde{\mathbf{u}}$ .
- $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as one of the following:

$$\begin{aligned} \text{bind-first: } \quad \tilde{\mathbf{t}} &= \langle \mathbf{D}_{out}(\text{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, f(\text{val}, \tilde{\mathbf{u}}(i))) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle, \\ \text{bind-second: } \quad \tilde{\mathbf{t}} &= \langle \mathbf{D}_{out}(\text{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, f(\tilde{\mathbf{u}}(i), \text{val})) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle, \end{aligned}$$

where  $f = \mathbf{f}(\text{op})$ .

The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

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  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

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.

#### 5979 4.3.8.4 apply: Matrix-BinaryOp variants[Scott: NEW CONTENT]

5980 Computes the transformation of the values of the stored elements of a matrix using a binary  
5981 operator and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the  
5982 first argument to the binary operator and stored elements of the matrix are passed as the second  
5983 argument. In the *bind-second* variant, the elements of the matrix are passed as the first argument  
5984 and the specified scalar value is passed as the second argument. The scalar can be passed either as  
5985 a non-opaque variable or as a GrB\_Scalar object.

#### 5986 C Syntax

```
5987 // bind-first + scalar value
5988 GrB_Info GrB_apply(GrB_Matrix      C,
5989                   const GrB_Matrix Mask,
5990                   const GrB_BinaryOp accum,
5991                   const GrB_BinaryOp op,
5992                   <type>           val,
5993                   const GrB_Matrix A,
5994                   const GrB_Descriptor desc);
```

```
5995 // bind-first + GraphBLAS scalar
5996 GrB_Info GrB_apply(GrB_Matrix      C,
5997                   const GrB_Matrix Mask,
5998                   const GrB_BinaryOp accum,
5999                   const GrB_BinaryOp op,
6000                   const GrB_Scalar s,
6001                   const GrB_Matrix A,
6002                   const GrB_Descriptor desc);
```

```
6003 // bind-second + scalar value
6004 GrB_Info GrB_apply(GrB_Matrix      C,
6005                   const GrB_Matrix Mask,
6006                   const GrB_BinaryOp accum,
6007                   const GrB_BinaryOp op,
6008                   const GrB_Matrix A,
6009                   <type>           val,
6010                   const GrB_Descriptor desc);
```

```
6011 // bind-second + GraphBLAS scalar
6012 GrB_Info GrB_apply(GrB_Matrix      C,
6013                   const GrB_Matrix Mask,
6014                   const GrB_BinaryOp accum,
6015                   const GrB_BinaryOp op,
6016                   const GrB_Matrix A,
```

```

6017         const GrB_Scalar      s,
6018         const GrB_Descriptor desc);

```

## 6019 Parameters

6020     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
6021     that may be accumulated with the result of the apply operation. On output, the  
6022     matrix holds the results of the operation.

6023     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
6024     stored into the output matrix C. The mask dimensions must match those of the  
6025     matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
6026     of the Mask matrix must be of type `bool` or any of the predefined “built-in” types  
6027     in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
6028     dimensions of C), `GrB_NULL` should be specified.

6029     **accum** (IN) An optional binary operator used for accumulating entries into existing C  
6030     entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
6031     specified.

6032     **op** (IN) A binary operator applied to each element of input matrix, A, with the element  
6033     of the input matrix used as the left-hand argument, and the scalar value, `val`, used  
6034     as the right-hand argument.

6035     **A** (IN) The GraphBLAS matrix whose elements are passed to the binary operator as  
6036     the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)  
6037     argument in the *bind-second* variant.

6038     **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)  
6039     argument in the *bind-first* variant, or the right-hand (second) argument in the  
6040     *bind-second* variant.

6041     **s** (IN) GraphBLAS scalar value that is passed to the binary operator as the left-hand  
6042     (first) argument in the *bind-first* variant, or the right-hand (second) argument in  
6043     the *bind-second* variant. It must not be empty.

6044     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
6045     should be specified. Non-default field/value pairs are listed as follows:

6046

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation ( <i>bind-second</i> variant only).
A	GrB_INP1	GrB_TRAN	Use transpose of A for the operation ( <i>bind-first</i> variant only).

## Return Values

GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.
GrB_PANIC	Unknown internal error.
GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call <code>GrB_error()</code> to access any error messages generated by the implementation.
GrB_OUT_OF_MEMORY	Not enough memory available for the operation.
GrB_UNINITIALIZED_OBJECT	One or more of the GraphBLAS objects has not been initialized by a call to <code>new</code> (or <code>Matrix_dup</code> for matrix parameters).
GrB_INDEX_OUT_OF_BOUNDS	A value in <code>row_indices</code> is greater than or equal to <code>nrows(A)</code> , or a value in <code>col_indices</code> is greater than or equal to <code>ncols(A)</code> . In non-blocking mode, this can be reported as an execution error.
GrB_DIMENSION_MISMATCH	Mask and C dimensions are incompatible, <code>nrows</code> $\neq$ <code>nrows(C)</code> , or <code>ncols</code> $\neq$ <code>ncols(C)</code> .
GrB_DOMAIN_MISMATCH	The domains of the various matrices and scalar are incompatible with the corresponding domains of the binary operator or accumulation operator, or the mask's domain is not compatible with <code>bool</code> (in the case where <code>desc[GrB_MASK].GrB_STRUCTURE</code> is not set).
GrB_EMPTY_OBJECT	The <code>GrB_Scalar s</code> used in the call is empty ( <code>nvals(s) = 0</code> ) and therefore a value cannot be passed to the binary operator.

## 6074 Description

6075 This variant of `GrB_apply` computes the result of applying a binary operator to the elements of a  
6076 GraphBLAS matrix each composed with a scalar constant, `val` or `s`:

6077                   bind-first:      $C = f(\text{val}, A)$  or  $C = f(s, A)$

6078                   bind-second:     $C = f(A, \text{val})$  or  $C = f(A, s)$ ,

6079 or if an optional binary accumulation operator ( $\odot$ ) is provided:

6080                   bind-first:      $C = C \odot f(\text{val}, A)$  or  $C = C \odot f(s, A)$

6081                   bind-second:     $C = C \odot f(A, \text{val})$  or  $C = C \odot f(A, s)$ .

6082 Logically, this operation occurs in three steps:

6083       **Setup** The internal matrices and mask used in the computation are formed and their domains  
6084               and dimensions are tested for compatibility.

6085       **Compute** The indicated computations are carried out.

6086       **Output** The result is written into the output matrix, possibly under control of a mask.

6087 Up to three argument matrices are used in the `GrB_apply` operation:

- 6088     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6089     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)
- 6090     3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6091 The argument scalar, matrices, binary operator and the accumulation operator (if provided) are  
6092 tested for domain compatibility as follows:

- 6093     1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
6094       must be from one of the pre-defined types of Table 3.2.
- 6095     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the binary operator.
- 6096     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
6097       of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the binary operator must be compatible with  
6098        $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 6099     4.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.
- 6100     5. If bind-first:
  - 6101       (a)  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the binary operator.

6102 (b) If the non-opaque scalar  $\text{val}$  is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$   
 6103 of the binary operator.

6104 (c) If the `GrB_Scalar`  $s$  is provided, then  $\mathbf{D}(s)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the  
 6105 binary operator.

6106 6. If `bind-second`:

6107 (a)  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.

6108 (b) If the non-opaque scalar  $\text{val}$  is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$   
 6109 of the binary operator.

6110 (c) If the `GrB_Scalar`  $s$  is provided, then  $\mathbf{D}(s)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the  
 6111 binary operator.

6112 Two domains are compatible with each other if values from one domain can be cast to values in  
 6113 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 6114 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 6115 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
 6116 error listed above is returned.

6117 From the argument matrices, the internal matrices, mask, and index arrays used in the computation  
 6118 are formed ( $\leftarrow$  denotes copy):

6119 1. Matrix  $\tilde{C} \leftarrow C$ .

6120 2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument `Mask` as follows:

6121 (a) If `Mask` = `GrB_NULL`, then  $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$   
 6122  $j < \mathbf{ncols}(C)\} \rangle$ .

6123 (b) If `Mask`  $\neq$  `GrB_NULL`,

6124 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
 6125  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,

6126 ii. Otherwise,  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
 6127  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .

6128 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{M} \leftarrow \neg \tilde{M}$ .

6129 3. Matrix  $\tilde{A}$  is computed from argument `A` as follows:

6130 `bind-first`:  $\tilde{A} \leftarrow \text{desc}[\text{GrB\_INP1}].\text{GrB\_TRAN} ? A^T : A$

6131 `bind-second`:  $\tilde{A} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? A^T : A$

6132 4. Scalar  $\tilde{s} \leftarrow s$  (`GraphBLAS` scalar case).

6133 The internal matrices and mask are checked for dimension compatibility. The following conditions  
 6134 must hold:

6135 1.  $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{M})$ .

6136 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .

6137 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .

6138 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

6139 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6140 error listed above is returned.

6141 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6142 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6143 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.

6144 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable  
6145 `val` with the same domain as  $\tilde{s}$  and set `val = val( $\tilde{s}$ )`.

6146 We are now ready to carry out the apply and any additional associated operations. We describe  
6147 this in terms of two intermediate matrices:

- 6148 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the binary operator to the input matrix  $\tilde{\mathbf{A}}$ .
- 6149 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6150 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as one of the following:

6151 bind-first:  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\mathbf{val}, \tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

6152 bind-second:  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j), \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

6153 where  $f = \mathbf{f}(\mathbf{op})$ .

6154 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6155 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6156 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6157 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6158 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6159 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$6160 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$6161 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$6162 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

6165 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

6166 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 6167 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 6168 mask which acts as a “write mask”.

- 6169 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
 6170 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6171 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6172 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 6173 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 6174 mask are unchanged:

$$6175 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6176 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 6177 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 6178 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
 6179 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 6180 sequence.

#### 6181 4.3.8.5 apply: Vector index unary operator variant[Scott: NEW CONTENT]

6182 Computes the transformation of the values of the stored elements of a vector using an index unary  
 6183 operator that is a function of the stored value, its location indices, and an user provided scalar  
 6184 value. The scalar can be passed either as a non-opaque variable or as a GrB\_Scalar object.

#### 6185 C Syntax

```
6186      GrB_Info GrB_apply(GrB_Vector          w,
6187                        const GrB_Vector     mask,
6188                        const GrB_BinaryOp    accum,
6189                        const GrB_IndexUnaryOp op,
6190                        const GrB_Vector     u,
6191                        <type>                val,
6192                        const GrB_Descriptor  desc);
```

```
6193      GrB_Info GrB_apply(GrB_Vector          w,
6194                        const GrB_Vector     mask,
6195                        const GrB_BinaryOp    accum,
6196                        const GrB_IndexUnaryOp op,
6197                        const GrB_Vector     u,
6198                        const GrB_Scalar     s,
6199                        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.

6231                   GrB\_PANIC   Unknown internal error.

6232           GrB\_INVALID\_OBJECT   This is returned in any execution mode whenever one of the  
6233                                   opaque GraphBLAS objects (input or output) is in an invalid  
6234                                   state caused by a previous execution error. Call `GrB_error()` to  
6235                                   access any error messages generated by the implementation.

6236           GrB\_OUT\_OF\_MEMORY   Not enough memory available for operation.

6237   GrB\_UNINITIALIZED\_OBJECT   One or more of the GraphBLAS objects has not been initialized  
6238                                   by a call to `new` (or another constructor).

6239   GrB\_DIMENSION\_MISMATCH   `mask`, `w` and/or `u` dimensions are incompatible.

6240   GrB\_DOMAIN\_MISMATCH   The domains of the various vectors are incompatible with the cor-  
6241                                   responding domains of the accumulation operator or index unary  
6242                                   operator, or the mask's domain is not compatible with `bool` (in  
6243                                   the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6244           GrB\_EMPTY\_OBJECT   The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and  
6245                                   therefore a value cannot be passed to the index unary operator.

## 6246   Description

6247   This variant of `GrB_apply` computes the result of applying an index unary operator to the elements  
6248   of a GraphBLAS vector each composed with the element's index and a scalar constant, `val` or `s`:

$$6249 \qquad w = f_i(u, \mathbf{ind}(u), 0, \text{val}) \text{ or } w = f_i(u, \mathbf{ind}(u), 0, s),$$

6250   or if an optional binary accumulation operator ( $\odot$ ) is provided:

$$6251 \qquad w = w \odot f_i(u, \mathbf{ind}(u), 0, \text{val}) \text{ or } w = w \odot f_i(u, \mathbf{ind}(u), 0, s).$$

6252   Logically, this operation occurs in three steps:

6253       **Setup**   The internal vectors and mask used in the computation are formed and their domains  
6254                   and dimensions are tested for compatibility.

6255       **Compute**   The indicated computations are carried out.

6256       **Output**   The result is written into the output vector, possibly under control of a mask.

6257   Up to three argument vectors are used in this `GrB_apply` operation:

- 6258       1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6259       2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)

6260 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6261 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)  
6262 are tested for domain compatibility as follows:

- 6263 1. If  $\mathbf{mask}$  is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
6264 must be from one of the pre-defined types of Table 3.2.
- 6265 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator.
- 6266 3. If  $\mathbf{accum}$  is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
6267 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator must be compatible  
6268 with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accumulation operator.
- 6269 4.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the index unary operator.
- 6270 5. If the non-opaque scalar  $\mathbf{val}$  is provided, then  $\mathbf{D}(\mathbf{val})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$  of  
6271 the index unary operator.
- 6272 6. 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 index  
6273 unary operator.

6274 Two domains are compatible with each other if values from one domain can be cast to values in  
6275 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6276 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6277 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
6278 error listed above is returned.

6279 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
6280 denotes copy):

- 6281 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 6282 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument  $\mathbf{mask}$  as follows:
  - 6283 (a) If  $\mathbf{mask} = \mathbf{GrB\_NULL}$ , then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 6284 (b) If  $\mathbf{mask} \neq \mathbf{GrB\_NULL}$ ,
    - 6285 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 6286 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
  - 6287 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 6288 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 6289 4. Scalar  $\tilde{s} \leftarrow \mathbf{s}$  (GraphBLAS scalar case).

6290 The internal vectors and masks are checked for dimension compatibility. The following conditions  
6291 must hold:

6292 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

6293 2.  $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$ .

6294 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6295 error listed above is returned.

6296 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6297 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6298 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6299 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided ( $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable `val`  
6300 with the same domain as  $\tilde{s}$  and set `val = val( $\tilde{s}$ )`.

6301 We are now ready to carry out the apply and any additional associated operations. We describe  
6302 this in terms of two intermediate vectors:

- 6303 •  $\tilde{\mathbf{t}}$ : The vector holding the result from applying the index unary operator to the input vector  
6304  $\tilde{\mathbf{u}}$ .
- 6305 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6306 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$6307 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f_i(\tilde{\mathbf{u}}(i), [i], 0, \text{val})) \mid i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,$$

6308 where  $f_i = \mathbf{f}(\text{op})$ .

6309 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6310 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 6311 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$6312 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6313 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
6314 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 6315 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 6316 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6317 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6318 \quad & \\ 6319 \end{aligned}$$

6320 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

6321 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
6322 using what is called a *standard vector mask and replace*. This is carried out under control of the  
6323 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[Scott: NEW CONTENT]

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.

6359       Mask (IN) An optional “write” mask that controls which results from this operation are  
6360       stored into the output matrix C. The mask dimensions must match those of the  
6361       matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
6362       of the Mask matrix must be of type **bool** or any of the predefined “built-in” types  
6363       in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6364       dimensions of C), GrB\_NULL should be specified.

6365       accum (IN) An optional binary operator used for accumulating entries into existing C  
6366       entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
6367       specified.

6368       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  
6369       to each element stored in the input matrix, A. It is a function of the stored element’s  
6370       value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6371       A (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-  
6372       ator.

6373       val (IN) An additional scalar value that is passed to the index unary operator.

6374       s (IN) An additional GraphBLAS scalar that is passed to the index unary operator.

6375       desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
6376       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.

## 6379   Return Values

6380       GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
6381       blocking mode, this indicates that the compatibility tests on di-  
6382       mensions and domains for the input arguments passed successfully.  
6383       Either way, output matrix C is ready to be used in the next method  
6384       of the sequence.

6385       GrB\_PANIC Unknown internal error.

6386       GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
6387       GraphBLAS objects (input or output) is in an invalid state caused

6388 by a previous execution error. Call `GrB_error()` to access any error  
 6389 messages generated by the implementation.

6390 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

6391 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 6392 a call to `new` (or another constructor).

6393 **GrB\_DIMENSION\_MISMATCH** `mask`, `w` and/or `u` dimensions are incompatible.

6394 **GrB\_DOMAIN\_MISMATCH** The domains of the various matrices are incompatible with the  
 6395 corresponding domains of the accumulation operator or index unary  
 6396 operator, or the mask's domain is not compatible with `bool` (in the  
 6397 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6398 **GrB\_EMPTY\_OBJECT** The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and  
 6399 therefore a value cannot be passed to the index unary operator.

## 6400 Description

6401 This variant of `GrB_apply` computes the result of applying a index unary operator to the elements  
 6402 of a GraphBLAS matrix each composed with the elements row and column indices, and a scalar  
 6403 constant, `val` or `s`:

$$6404 \quad C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s),$$

6405 or if an optional binary accumulation operator ( $\odot$ ) is provided:

$$6406 \quad C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s).$$

6407 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional  
 6408 indices, respectively.

6409 Logically, this operation occurs in three steps:

6410 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 6411 and dimensions are tested for compatibility.

6412 **Compute** The indicated computations are carried out.

6413 **Output** The result is written into the output matrix, possibly under control of a mask.

6414 Up to three argument matrices are used in the `GrB_apply` operation:

- 6415 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6416 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)

6417 3.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

6418 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)  
6419 are tested for domain compatibility as follows:

- 6420 1. If **Mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\mathbf{Mask})$   
6421 must be from one of the pre-defined types of Table 3.2.
- 6422 2.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator.
- 6423 3. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
6424 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator must be compatible  
6425 with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accumulation operator.
- 6426 4.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the index unary operator.
- 6427 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  
6428 the index unary operator.
- 6429 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  
6430 unary operator.

6431 Two domains are compatible with each other if values from one domain can be cast to values in  
6432 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6433 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6434 any compatibility rule above is violated, execution of **GrB\_apply** ends and the domain mismatch  
6435 error listed above is returned.

6436 From the argument matrices, the internal matrices, **mask**, and index arrays used in the computation  
6437 are formed ( $\leftarrow$  denotes copy):

- 6438 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 6439 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument **Mask** as follows:
  - 6440 (a) If **Mask** = **GrB\_NULL**, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
6441  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 6442 (b) If **Mask**  $\neq$  **GrB\_NULL**,
    - 6443 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
6444  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 6445 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
6446  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 6447 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 6448 3. Matrix  $\tilde{\mathbf{A}}$  is computed from argument **A** as follows:
  - 6449  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc[GrB_INP0].GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$
- 6450 4. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).



6451 The internal matrices and mask are checked for dimension compatibility. The following conditions  
6452 must hold:

- 6453 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 6454 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 6455 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
- 6456 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

6457 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6458 error listed above is returned.

6459 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6460 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6461 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6462 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable  
6463 `val` with the same domain as  $\tilde{s}$  and set `val = val( $\tilde{s}$ )`.

6464 We are now ready to carry out the apply and any additional associated operations. We describe  
6465 this in terms of two intermediate matrices:

- 6466 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the index unary operator to the input matrix  
6467  $\tilde{\mathbf{A}}$ .
- 6468 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6469 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$6470 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f_i(\tilde{\mathbf{A}}(i, j), i, j, \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

6471 where  $f_i = \mathbf{f}(\mathbf{op})$ .

6472 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6473 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6474 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6475 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6476 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6477 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 6478 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6479 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6480 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6481 \quad & \\ 6482 \end{aligned}$$

6483 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

6484 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
6485 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
6486 mask which acts as a “write mask”.

- 6487 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
6488 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6489 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6490 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
6491 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
6492 mask are unchanged:

$$6493 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6494 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
6495 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
6496 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
6497 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
6498 sequence.

#### 6499 4.3.9 select:

6500 Apply a select operator to the stored elements of an object to determine whether or not to keep  
6501 them.

##### 6502 4.3.9.1 select: Vector variant[Scott: NEW CONTENT]

6503 Apply a select operator (an index unary operator) to the elements of a vector.

#### 6504 C Syntax

```
6505 // scalar value variant
6506 GrB_Info GrB_select(GrB_Vector          w,
6507                    const GrB_Vector      mask,
6508                    const GrB_BinaryOp    accum,
6509                    const GrB_IndexUnaryOp op,
6510                    const GrB_Vector      u,
6511                    <type>                val,
6512                    const GrB_Descriptor   desc);
6513
6514 // GraphBLAS scalar variant
6515 GrB_Info GrB_select(GrB_Vector          w,
6516                    const GrB_Vector      mask,
```

```

6517         const GrB_BinaryOp      accum,
6518         const GrB_IndexUnaryOp  op,
6519         const GrB_Vector        u,
6520         const GrB_Scalar        s,
6521         const GrB_Descriptor    desc);
6522

```

## 6523 Parameters

6524     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
6525     that may be accumulated with the result of the select operation. On output, this  
6526     vector holds the results of the operation.

6527     **mask** (IN) An optional “write” mask that controls which results from this operation are  
6528     stored into the output vector **w**. The mask dimensions must match those of the  
6529     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6530     of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
6531     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6532     dimensions of **w**), **GrB\_NULL** should be specified.

6533     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
6534     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6535     specified.

6536     **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  
6537     to each element stored in the input vector, **u**. It is a function of the stored element’s  
6538     value, its location index, and a user supplied scalar value (either **s** or **val**).

6539     **u** (IN) The GraphBLAS vector whose elements are passed to the index unary oper-  
6540     ator.

6541     **val** (IN) An additional scalar value that is passed to the index unary operator.

6542     **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must  
6543     not be empty.

6544     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
6545     should be specified. Non-default field/value pairs are listed as follows:

6546

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> .

6547

## 6548 Return Values

6549           **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 6550                           blocking mode, this indicates that the compatibility tests on di-  
 6551                           mensions and domains for the input arguments passed success-  
 6552                           fully. Either way, output vector **w** is ready to be used in the next  
 6553                           method of the sequence.

6554           **GrB\_PANIC** Unknown internal error.

6555           **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
 6556                           opaque GraphBLAS objects (input or output) is in an invalid  
 6557                           state caused by a previous execution error. Call **GrB\_error()** to  
 6558                           access any error messages generated by the implementation.

6559           **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

6560           **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized  
 6561                           by a call to one of its constructors.

6562           **GrB\_DIMENSION\_MISMATCH** **mask**, **w** and/or **u** dimensions are incompatible.

6563           **GrB\_DOMAIN\_MISMATCH** The domains of the various vectors are incompatible with the cor-  
 6564                           responding domains of the accumulation operator or index unary  
 6565                           operator, or the mask's domain is not compatible with **bool** (in  
 6566                           the case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

6567           **GrB\_EMPTY\_OBJECT** The **GrB\_Scalar s** used in the call is empty (**nvals(s) = 0**) and  
 6568                           therefore a value cannot be passed to the index unary operator.

## 6569 Description

6570 This variant of **GrB\_select** computes the result of applying a index unary operator to select the  
 6571 elements of the input GraphBLAS vector. The operator takes, as input, the value of each stored  
 6572 element, along with the element's index and a scalar constant – either **val** or **s**. The corresponding  
 6573 element of the input vector is selected (kept) if the function evaluates to **true** when cast to **bool**.  
 6574 This acts like a functional mask on the input vector as follows:

$$6575 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle,$$

$$6576 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle.$$

6577 Correspondingly, if a **GrB\_Scalar s**, is provided:

$$6578 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle,$$

$$6579 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle.$$

6580 Logically, this operation occurs in three steps:

6581     **Setup** The internal vectors and mask used in the computation are formed and their domains  
6582             and dimensions are tested for compatibility.

6583     **Compute** The indicated computations are carried out.

6584     **Output** The result is written into the output vector, possibly under control of a mask.

6585 Up to three argument vectors are used in this `GrB_select` operation:

- 6586     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$   
6587     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)  
6588     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6589 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)  
6590 are tested for domain compatibility as follows:

- 6591     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
6592         must be from one of the pre-defined types of Table 3.2.  
6593     2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .  
6594     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
6595         of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accu-  
6596         mulation operator.  
6597     4.  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator must be from one of the pre-defined types of Table 3.2;  
6598         i.e., castable to `bool`.  
6599     5.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the index unary operator.  
6600     6.  $\mathbf{D}(\mathbf{val})$  or  $\mathbf{D}(\mathbf{s})$ , depending on the signature of the method, must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$   
6601         of the index unary operator.

6602 Two domains are compatible with each other if values from one domain can be cast to values in  
6603 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6604 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6605 any compatibility rule above is violated, execution of `GrB_select` ends and the domain mismatch  
6606 error listed above is returned.

6607 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
6608 denotes copy):

- 6609     1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .  
6610     2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:

6611 (a) If  $\text{mask} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .  
6612 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,  
6613 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,  
6614 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .  
6615 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .  
6616 3. Vector  $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$ .  
6617 4. Scalar  $\widetilde{s} \leftarrow s$  (GrB\_Scalar version only).

6618 The internal vectors and masks are checked for dimension compatibility. The following conditions  
6619 must hold:

- 6620 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 6621 2.  $\text{size}(\widetilde{\mathbf{u}}) = \text{size}(\widetilde{\mathbf{w}})$ .

6622 If any compatibility rule above is violated, execution of `GrB_select` ends and the dimension mismatch  
6623 error listed above is returned.

6624 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6625 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6626 If an empty `GrB_Scalar`  $\widetilde{s}$  is provided (i.e.,  $\text{nvals}(\widetilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6627 If a non-empty `GrB_Scalar`,  $\widetilde{s}$ , is provided (i.e.,  $\text{nvals}(\widetilde{s}) = 1$ ), we then create an internal variable  
6628 `val` with the same domain as  $\widetilde{s}$  and set  $\text{val} = \text{val}(\widetilde{s})$ .

6629 We are now ready to carry out the `select` and any additional associated operations. We describe  
6630 this in terms of two intermediate vectors:

- 6631 •  $\widetilde{\mathbf{t}}$ : The vector holding the result from applying the index unary operator to the input vector  
6632  $\widetilde{\mathbf{u}}$ .
- 6633 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6634 The intermediate vector,  $\widetilde{\mathbf{t}}$ , is created as follows:

$$6635 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{u}}), \{(i, \widetilde{\mathbf{u}}(i), : i \in \text{ind}(\widetilde{\mathbf{u}}) \wedge (\text{bool})f_i(\widetilde{\mathbf{u}}(i), i, 0, \text{val}) = \text{true})\} \rangle,$$

6636 where  $f_i = \mathbf{f}(\text{op})$ .

6637 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6638 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$ .
- 6639 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{z}}$  is defined as

$$6640 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ , using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a “write mask”.

- If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

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

#### 4.3.9.2 select: Matrix variant[Scott: NEW CONTENT]

Apply a select operator (an index unary operator) to the elements of a matrix.

#### C Syntax

```
// scalar value variant
GrB_Info GrB_select(GrB_Matrix          C,
                   const GrB_Matrix      Mask,
                   const GrB_BinaryOp     accum,
                   const GrB_IndexUnaryOp op,
                   const GrB_Matrix      A,
                   <type>                 val,
                   const GrB_Descriptor   desc);
```

```

6676 // GraphBLAS scalar variant
6677 GrB_Info GrB_select(GrB_Matrix          C,
6678                    const GrB_Matrix     Mask,
6679                    const GrB_BinaryOp   accum,
6680                    const GrB_IndexUnaryOp op,
6681                    const GrB_Matrix     A,
6682                    const GrB_Scalar     s,
6683                    const GrB_Descriptor  desc);

```

## 6684 Parameters

6685     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
6686     that may be accumulated with the result of the select operation. On output, the  
6687     matrix holds the results of the operation.

6688     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
6689     stored into the output matrix **C**. The mask dimensions must match those of the  
6690     matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6691     of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
6692     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6693     dimensions of **C**), **GrB\_NULL** should be specified.

6694     **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
6695     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6696     specified.

6697     **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  
6698     to each element stored in the input matrix, **A**. It is a function of the stored element’s  
6699     value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6700     **A** (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-  
6701     ator.

6702     **val** (IN) An additional scalar value that is passed to the index unary operator.

6703     **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must  
6704     not be empty.

6705     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
6706     should be specified. Non-default field/value pairs are listed as follows:

6707



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:

6737  $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$ , or

6738  $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$ .

6739 Correspondingly, if a GrB\_Scalar,  $s$ , is provided:

6740  $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$ , or

6741  $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$ .

6742 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional  
6743 indices, respectively.

6744 Logically, this operation occurs in three steps:

6745     **Setup** The internal matrices and mask used in the computation are formed and their domains  
6746             and dimensions are tested for compatibility.

6747     **Compute** The indicated computations are carried out.

6748     **Output** The result is written into the output matrix, possibly under control of a mask.

6749 Up to three argument matrices are used in the GrB\_select operation:

- 6750 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6751 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)
- 6752 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6753 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)  
6754 are tested for domain compatibility as follows:

- 6755 1. If **Mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{Mask})$   
6756     must be from one of the pre-defined types of Table 3.2.
- 6757 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$ .
- 6758 3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
6759     of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
6760     mulation operator.
- 6761 4.  $\mathbf{D}_{out}(\text{op})$  of the index unary operator must be from one of the pre-defined types of Table 3.2;  
6762     i.e., castable to **bool**.
- 6763 5.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the index unary operator.
- 6764 6.  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(s)$ , depending on the signature of the method, must be compatible with  $\mathbf{D}_{in_2}(\text{op})$   
6765     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:

- 6800 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the index unary operator to the input matrix  
6801  $\tilde{\mathbf{A}}$ .
- 6802 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6803 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$6804 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \\ \{(i, j, \tilde{\mathbf{A}}(i, j) : i, j \in \mathbf{ind}(\tilde{\mathbf{A}}) \wedge (\text{bool})f_i(\tilde{\mathbf{A}}(i, j), i, j, \text{val}) = \text{true})\},$$

6805 where  $f_i = \mathbf{f}(\text{op})$ .

6806 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6807 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6808 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6809 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\}\rangle.$$

6810 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6811 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$6812 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6813 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6814 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6815 \quad 6816$$

6817 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

6818 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
6819 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
6820 mask which acts as a “write mask”.

- 6821 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
6822 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6823 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6824 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
6825 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
6826 mask are unchanged:

$$6827 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6828 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
6829 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
6830 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
6831 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
6832 sequence.

### 6833 4.3.10 reduce: Perform a reduction across the elements of an object

6834 Computes the reduction of the values of the elements of a vector or matrix.

#### 6835 4.3.10.1 reduce: Standard matrix to vector variant

6836 This performs a reduction across rows of a matrix to produce a vector. If reduction down columns  
6837 is desired, the input matrix should be transposed using the descriptor.

### 6838 C Syntax

```
6839         GrB_Info GrB_reduce(GrB_Vector          w,  
6840                             const GrB_Vector    mask,  
6841                             const GrB_BinaryOp   accum,  
6842                             const GrB_Monoid     op,  
6843                             const GrB_Matrix    A,  
6844                             const GrB_Descriptor desc);  
6845  
6846         GrB_Info GrB_reduce(GrB_Vector          w,  
6847                             const GrB_Vector    mask,  
6848                             const GrB_BinaryOp   accum,  
6849                             const GrB_BinaryOp   op,  
6850                             const GrB_Matrix    A,  
6851                             const GrB_Descriptor desc);
```

### 6852 Parameters

6853 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
6854 that may be accumulated with the result of the reduction operation. On output,  
6855 this vector holds the results of the operation.

6856 **mask** (IN) An optional “write” mask that controls which results from this operation are  
6857 stored into the output vector **w**. The mask dimensions must match those of the  
6858 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6859 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
6860 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6861 dimensions of **w**), **GrB\_NULL** should be specified.

6862 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
6863 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6864 specified.

6865 **op** (IN) The monoid or binary operator used in the element-wise reduction operation.  
6866 Depending on which type is passed, the following defines the binary operator with  
6867 one domain,  $F_b = \langle D, D, D, \oplus \rangle$ , that is used:

6868 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .  
6869 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  
6870 monoid is ignored.

6871 If `op` is a `GrB_BinaryOp`, then all its domains must be the same. Furthermore, in  
6872 both cases  $\odot(\text{op})$  must be commutative and associative. Otherwise, the outcome  
6873 of the operation is undefined.

6874 **A** (IN) The GraphBLAS matrix on which reduction will be performed.

6875 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
6876 should be specified. Non-default field/value pairs are listed as follows:  
6877

Param	Field	Value	Description
<code>w</code>	<code>GrB_OUTP</code>	<code>GrB_REPLACE</code>	Output vector <code>w</code> is cleared (all elements removed) before the result is stored in it.
<code>mask</code>	<code>GrB_MASK</code>	<code>GrB_STRUCTURE</code>	The write mask is constructed from the structure (pattern of stored values) of the input <code>mask</code> vector. The stored values are not examined.
<code>mask</code>	<code>GrB_MASK</code>	<code>GrB_COMP</code>	Use the complement of <code>mask</code> .
<code>A</code>	<code>GrB_INP0</code>	<code>GrB_TRAN</code>	Use transpose of <code>A</code> for the operation.

## 6879 Return Values

6880 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
6881 blocking mode, this indicates that the compatibility tests on di-  
6882 mensions and domains for the input arguments passed successfully.  
6883 Either way, output vector `w` is ready to be used in the next method  
6884 of the sequence.

6885 **GrB\_PANIC** Unknown internal error.

6886 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
6887 GraphBLAS objects (input or output) is in an invalid state caused  
6888 by a previous execution error. Call `GrB_error()` to access any error  
6889 messages generated by the implementation.

6890 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

6891 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
6892 a call to `new` (or `dup` for vector parameters).

6893 **GrB\_DIMENSION\_MISMATCH** `mask`, `w` and/or `u` dimensions are incompatible.

6894 **GrB\_DOMAIN\_MISMATCH** Either the domains of the various vectors and matrices are incom-  
6895 patible with the corresponding domains of the accumulation oper-  
6896 ator or reduce function, or the domains of the GraphBLAS binary

6897 operator `op` are not all the same, or the mask's domain is not com-  
6898 patible with `bool` (in the case where `desc[GrB_MASK].GrB_STRUCTURE`  
6899 is not set).

## 6900 Description

6901 This variant of `GrB_reduce` computes the result of performing a reduction across each of the rows  
6902 of an input matrix:  $w(i) = \bigoplus A(i, :) \forall i$ ; or, if an optional binary accumulation operator is provided,  
6903  $w(i) = w(i) \odot (\bigoplus A(i, :)) \forall i$ , where  $\bigoplus = \odot(F_b)$  and  $\odot = \odot(\text{accum})$ .

6904 Logically, this operation occurs in three steps:

6905     **Setup** The internal vector, matrix and mask used in the computation are formed and their  
6906 domains and dimensions are tested for compatibility.

6907 **Compute** The indicated computations are carried out.

6908 **Output** The result is written into the output vector, possibly under control of a mask.

6909 Up to two vector and one matrix argument are used in this `GrB_reduce` operation:

- 6910 1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6911 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 6912 3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6913 The argument vector, matrix, reduction operator and accumulation operator (if provided) are tested  
6914 for domain compatibility as follows:

- 6915 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
6916 must be from one of the pre-defined types of Table 3.2.
- 6917 2.  $\mathbf{D}(w)$  must be compatible with the domain of the reduction binary operator,  $\mathbf{D}(F_b)$ .
- 6918 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
6919 of the accumulation operator and  $\mathbf{D}(F_b)$ , must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
6920 mulation operator.
- 6921 4.  $\mathbf{D}(A)$  must be compatible with the domain of the binary reduction operator,  $\mathbf{D}(F_b)$ .

6922 Two domains are compatible with each other if values from one domain can be cast to values in  
6923 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6924 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6925 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch  
6926 error listed above is returned.

6927 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
6928 denotes copy):

- 6929 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 6930 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
- 6931 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
- 6932 (b) If `mask  $\neq$  GrB_NULL`,
- 6933 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
- 6934 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
- 6935 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 6936 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

6937 The internal vectors and masks are checked for dimension compatibility. The following conditions  
6938 must hold:

- 6939 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 6940 2.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .

6941 If any compatibility rule above is violated, execution of `GrB_reduce` ends and the dimension mis-  
6942 match error listed above is returned.

6943 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6944 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6945 We carry out the reduce and any additional associated operations. We describe this in terms of  
6946 two intermediate vectors:

- 6947 •  $\tilde{\mathbf{t}}$ : The vector holding the result from reducing along the rows of input matrix  $\tilde{\mathbf{A}}$ .
- 6948 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6949 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$6950 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, t_i) : \mathbf{ind}(\mathbf{A}(i, :)) \neq \emptyset\} \rangle.$$

6951 The value of each of its elements is computed by

$$6952 \quad t_i = \bigoplus_{j \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :))} \tilde{\mathbf{A}}(i, j),$$

6953 where  $\bigoplus = \odot(F_b)$ .

6954 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6955 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .



- If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ , using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a “write mask”.

- If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

- If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

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

#### 4.3.10.2 reduce: Vector-scalar variant[Scott: NEW CONTENT]

Reduce all stored values into a single scalar.

### C Syntax

```
// scalar value + monoid (only)
GrB_Info GrB_reduce(<type>          *val,
                    const GrB_BinaryOp accum,
                    const GrB_Monoid  op,
                    const GrB_Vector  u,
```

```

6989             const GrB_Descriptor desc);
6990
6991 // GraphBLAS Scalar + monoid
6992 GrB_Info GrB_reduce(GrB_Scalar      s,
6993                    const GrB_BinaryOp accum,
6994                    const GrB_Monoid op,
6995                    const GrB_Vector u,
6996                    const GrB_Descriptor desc);
6997
6998 // GraphBLAS Scalar + binary operator
6999 GrB_Info GrB_reduce(GrB_Scalar      s,
7000                    const GrB_BinaryOp accum,
7001                    const GrB_BinaryOp op,
7002                    const GrB_Vector u,
7003                    const GrB_Descriptor desc);

```

## 7004 Parameters

7005 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides  
7006 a value that may be accumulated (optionally) with the result of the reduction  
7007 operation. On output, this scalar holds the results of the operation.

7008 **accum** (IN) An optional binary operator used for accumulating entries into an exist-  
7009 ing scalar (**s** or **val**) value. If assignment rather than accumulation is desired,  
7010 **GrB\_NULL** should be specified.

7011 **op** (IN) The monoid ( $M = \langle D, \oplus, 0 \rangle$ ) or binary operator ( $F_b = \langle D, D, D, \oplus \rangle$ ) used in  
7012 the reduction operation. The  $\oplus$  operator must be commutative and associative;  
7013 otherwise, the outcome of the operation is undefined.

7014 **u** (IN) The GraphBLAS vector on which reduction will be performed.

7015 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
7016 should be specified. Non-default field/value pairs are listed as follows:

7018 Param	Field	Value	Description
------------	-------	-------	-------------

7019 *Note:* This argument is defined for consistency with the other GraphBLAS opera-  
7020 tions. There are currently no non-default field/value pairs that can be set for this  
7021 operation.

## 7022 Return Values

7023 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
7024 cessfully, and the output scalar (**s** or **val**) is ready to be used in the  
7025 next method of the sequence.

7026	<b>GrB_PANIC</b>	Unknown internal error.
7027	<b>GrB_INVALID_OBJECT</b>	This is returned in any execution mode whenever one of the opaque
7028		GraphBLAS objects (input or output) is in an invalid state caused
7029		by a previous execution error. Call <b>GrB_error()</b> to access any error
7030		messages generated by the implementation.
7031	<b>GrB_OUT_OF_MEMORY</b>	Not enough memory available for the operation.
7032	<b>GrB_UNINITIALIZED_OBJECT</b>	One or more of the GraphBLAS objects has not been initialized by
7033		a call to a respective constructor.
7034	<b>GrB_NULL_POINTER</b>	val pointer is NULL.
7035	<b>GrB_DOMAIN_MISMATCH</b>	The domains of input and output arguments are incompatible with
7036		the corresponding domains of the accumulation operator, or reduce
7037		operator.

## 7038 Description

This variant of **GrB\_reduce** computes the result of performing a reduction across all of the stored elements of an input vector storing the result into either **s** or **val**. This corresponds to (shown here for the scalar value case only):

$$\text{val} = \begin{cases} \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i), & \text{or} \\ \text{val} \odot \left[ \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7039 where  $\bigoplus = \odot(\text{op})$  and  $\odot = \odot(\text{accum})$ .

7040 Logically, this operation occurs in three steps:

7041 **Setup** The internal vector used in the computation is formed and its domain is tested for  
7042 compatibility.

7043 **Compute** The indicated computations are carried out.

7044 **Output** The result is written into the output scalar.

7045 One vector argument is used in this **GrB\_reduce** operation:

- 7046 1.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

7047 The output scalar, argument vector, reduction operator and accumulation operator (if provided)  
7048 are tested for domain compatibility as follows:

- 7049 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  
7050  $\mathbf{D}_{in_1}(\text{op})$  and  $\mathbf{D}_{in_2}(\text{op})$  from  $F_b$ ).

- 7051 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  
7052  $\mathbf{D}_{out}(\text{accum})$  of the accumulation operator, and  $\mathbf{D}(\text{op})$  from  $M$  (or  $\mathbf{D}_{out}(\text{op})$  from  $F_b$ ) must  
7053 be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 7054 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  $F_b$ ).

7055 Two domains are compatible with each other if values from one domain can be cast to values in  
7056 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
7057 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
7058 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch  
7059 error listed above is returned.

7060 The number of values stored in the input, `u`, is checked. If there are no stored values in `u`, then one  
7061 of the following occurs depending on the output variant:

$$7062 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if } \text{accum} = \text{GrB\_NULL}, \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7063 or

$$7064 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if } \text{accum} = \text{GrB\_NULL}, \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7065 where  $\mathbf{0}(\text{op})$  is the identity of the monoid. The operation returns immediately with `GrB_SUCCESS`.

7066 For all other cases, the internal vector and scalar used in the computation is formed ( $\leftarrow$  denotes  
7067 copy):

- 7068 1. Vector  $\tilde{\mathbf{u}} \leftarrow \text{u}$ .
- 7069 2. Scalar  $\tilde{s} \leftarrow \text{s}$  (GraphBLAS scalar case).

7070 We are now ready to carry out the reduction and any additional associated operations. An inter-  
7071 mediate scalar result  $t$  is computed as follows:

$$7072 \quad t = \bigoplus_{i \in \text{ind}(\tilde{\mathbf{u}})} \tilde{\mathbf{u}}(i),$$

7073 where  $\oplus = \odot(\text{op})$ .

7074 The final reduction value is computed as follows:

$$7075 \quad \mathbf{L}(\text{s}) \leftarrow \begin{cases} \{t\}, & \text{when } \text{accum} = \text{GrB\_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\text{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7076 or

$$7077 \quad \text{val} \leftarrow \begin{cases} t, & \text{when } \text{accum} = \text{GrB\_NULL, or} \\ \text{val} \odot t, & \text{otherwise;} \end{cases}$$

7078 In both GrB\_BLOCKING and GrB\_NONBLOCKING modes, the method exits with return value  
7079 GrB\_SUCCESS and the new contents of the output scalar is as defined above.

#### 7080 4.3.10.3 reduce: Matrix-scalar variant[Scott: NEW CONTENT]

7081 Reduce all stored values into a single scalar.

### 7082 C Syntax

```
7083 // scalar value + monoid (only)
7084 GrB_Info GrB_reduce(<type>          *val,
7085                    const GrB_BinaryOp accum,
7086                    const GrB_Monoid   op,
7087                    const GrB_Matrix   A,
7088                    const GrB_Descriptor desc);
7089
7090 // GraphBLAS Scalar + monoid
7091 GrB_Info GrB_reduce(GrB_Scalar      s,
7092                    const GrB_BinaryOp accum,
7093                    const GrB_Monoid   op,
7094                    const GrB_Matrix   A,
7095                    const GrB_Descriptor desc);
7096
7097 // GraphBLAS Scalar + binary operator
7098 GrB_Info GrB_reduce(GrB_Scalar      s,
7099                    const GrB_BinaryOp accum,
7100                    const GrB_BinaryOp op,
7101                    const GrB_Matrix   A,
7102                    const GrB_Descriptor desc);
```

### 7103 Parameters

7104 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides  
7105 a value that may be accumulated (optionally) with the result of the reduction  
7106 operation. On output, this scalar holds the results of the operation.

7107 **accum** (IN) An optional binary operator used for accumulating entries into existing (**s** or  
7108 **val**) value. If assignment rather than accumulation is desired, GrB\_NULL should  
7109 be specified.

7110 **op** (IN) The monoid ( $M = \langle D, \oplus, 0 \rangle$ ) or binary operator ( $F_b = \langle D, D, D, \oplus \rangle$ ) used in  
7111 the reduction operation. The  $\oplus$  operator must be commutative and associative;  
7112 otherwise, the outcome of the operation is undefined.

7113 **A** (IN) The GraphBLAS matrix on which the reduction will be performed.

7114 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 7115 should be specified. Non-default field/value pairs are listed as follows:  
 7116

7117	Param	Field	Value	Description
------	-------	-------	-------	-------------

7118 *Note:* This argument is defined for consistency with the other GraphBLAS opera-  
 7119 tions. There are currently no non-default field/value pairs that can be set for this  
 7120 operation.

## 7121 Return Values

7122 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
 7123 cessfully, and the output scalar (s or val) is ready to be used in the  
 7124 next method of the sequence.

7125 GrB\_PANIC Unknown internal error.

7126 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 7127 GraphBLAS objects (input or output) is in an invalid state caused  
 7128 by a previous execution error. Call GrB\_error() to access any error  
 7129 messages generated by the implementation.

7130 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

7131 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
 7132 a call to a respective constructor.

7133 GrB\_NULL\_POINTER val pointer is NULL.

7134 GrB\_DOMAIN\_MISMATCH The domains of input and output arguments are incompatible with  
 7135 the corresponding domains of the accumulation operator, or reduce  
 7136 operator.

## 7137 Description

This variant of GrB\_reduce computes the result of performing a reduction across all of the stored elements of an input matrix storing the result into either s or val. This corresponds to (shown here for the scalar value case only):

$$\text{val} = \begin{cases} \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j), & \text{or} \\ \text{val} \odot \left[ \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7138 where  $\bigoplus = \odot(\text{op})$  and  $\odot = \odot(\text{accum})$ .

7139 Logically, this operation occurs in three steps:

7140       **Setup** The internal matrix used in the computation is formed and its domain is tested for  
 7141       compatibility.

7142       **Compute** The indicated computations are carried out.

7143       **Output** The result is written into the output scalar.

7144   One matrix argument is used in this GrB\_reduce operation:

7145       1.  $A = \langle \mathbf{D}(A), \mathbf{size}(A), \mathbf{L}(A) = \{(i, j, A_{i,j})\} \rangle$

7146   The output scalar, argument matrix, reduction operator and accumulation operator (if provided)  
 7147   are tested for domain compatibility as follows:

7148       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  
 7149        $\mathbf{D}_{in_1}(\text{op})$  and  $\mathbf{D}_{in_2}(\text{op})$  from  $F_b$ ).

7150       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  
 7151        $\mathbf{D}_{out}(\text{accum})$  of the accumulation operator, and  $\mathbf{D}(\text{op})$  from  $M$  (or  $\mathbf{D}_{out}(\text{op})$  from  $F_b$ ) must  
 7152       be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.

7153       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  $F_b$ ).

7154   Two domains are compatible with each other if values from one domain can be cast to values in  
 7155   the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 7156   compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 7157   any compatibility rule above is violated, execution of GrB\_reduce ends and the domain mismatch  
 7158   error listed above is returned.

7159   The number of values stored in the input,  $A$ , is checked. If there are no stored values in  $A$ , then  
 7160   one of the following occurs depending on the output variant:

$$7161 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if accum = GrB\_NULL,} \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7162   or

$$7163 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if accum = GrB\_NULL,} \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7164   where  $\mathbf{0}(\text{op})$  is the identity of the monoid. The operation returns immediately with GrB\_SUCCESS.

7165   For all other cases, the internal matrix and scalar used in the computation is formed ( $\leftarrow$  denotes  
 7166   copy):

7167       1. Matrix  $\tilde{A} \leftarrow A$ .

7168       2. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).

7169 We are now ready to carry out the reduce and any additional associated operations. An intermediate  
 7170 scalar result  $t$  is computed as follows:

$$7171 \quad t = \bigoplus_{(i,j) \in \text{ind}(\tilde{\mathbf{A}})} \tilde{\mathbf{A}}(i,j),$$

7172 where  $\oplus = \odot(\text{op})$ .

7173 The final reduction value is computed as follows:

$$7174 \quad \mathbf{L}(\mathbf{s}) \leftarrow \begin{cases} \{t\}, & \text{when accum} = \text{GrB\_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\mathbf{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7175 or

$$7176 \quad \mathbf{val} \leftarrow \begin{cases} t, & \text{when accum} = \text{GrB\_NULL, or} \\ \mathbf{val} \odot t, & \text{otherwise;} \end{cases}$$

7177 In both GrB\_BLOCKING and GrB\_NONBLOCKING modes, the method exits with return value  
 7178 GrB\_SUCCESS and the new contents of the output scalar is as defined above.

#### 7179 4.3.11 transpose: Transpose rows and columns of a matrix

7180 This version computes a new matrix that is the transpose of the source matrix.

#### 7181 C Syntax

```
7182      GrB_Info GrB_transpose(GrB_Matrix      C,
7183                           const GrB_Matrix Mask,
7184                           const GrB_BinaryOp accum,
7185                           const GrB_Matrix A,
7186                           const GrB_Descriptor desc);
```

#### 7187 Parameters

7188 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
 7189 that may be accumulated with the result of the transpose operation. On output,  
 7190 the matrix holds the results of the operation.

7191 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
 7192 stored into the output matrix C. The mask dimensions must match those of the  
 7193 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
 7194 of the Mask matrix must be of type bool or any of the predefined “built-in” types  
 7195 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
 7196 dimensions of C), GrB\_NULL should be specified.



7197        **accum** (IN) An optional binary operator used for accumulating entries into existing C  
7198                entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
7199                specified.

7200        **A** (IN) The GraphBLAS matrix on which transposition will be performed.

7201        **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
7202                should be specified. Non-default field/value pairs are listed as follows:  
7203

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.

## 7205 Return Values

7206        **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
7207                blocking mode, this indicates that the compatibility tests on di-  
7208                mensions and domains for the input arguments passed successfully.  
7209                Either way, output matrix C is ready to be used in the next method  
7210                of the sequence.

7211        **GrB\_PANIC** Unknown internal error.

7212        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
7213                GraphBLAS objects (input or output) is in an invalid state caused  
7214                by a previous execution error. Call **GrB\_error()** to access any error  
7215                messages generated by the implementation.

7216        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

7217        **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
7218                a call to **new** (or **Matrix\_dup** for matrix parameters).

7219        **GrB\_DIMENSION\_MISMATCH** mask, C and/or A dimensions are incompatible.

7220        **GrB\_DOMAIN\_MISMATCH** The domains of the various matrices are incompatible with the cor-  
7221                responding domains of the accumulation operator, or the mask's do-  
7222                main is not compatible with **bool** (in the case where **desc[GrB\_MASK].GrB\_STRUCTURE**  
7223                is not set).

## 7224 Description

7225 GrB\_transpose computes the result of performing a transpose of the input matrix:  $C = A^T$ ; or, if an  
 7226 optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot A^T$ . We note that the input matrix  
 7227 A can itself be optionally transposed before the operation, which would cause either an assignment  
 7228 from A to C or an accumulation of A into C.

7229 Logically, this operation occurs in three steps:

7230     **Setup** The internal matrix and mask used in the computation are formed and their domains  
 7231             and dimensions are tested for compatibility.

7232     **Compute** The indicated computations are carried out.

7233     **Output** The result is written into the output matrix, possibly under control of a mask.

7234 Up to three matrix arguments are used in this GrB\_transpose operation:

- 7235     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7236     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)
- 7237     3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

7238 The argument matrices and accumulation operator (if provided) are tested for domain compatibility  
 7239 as follows:

- 7240     1. If Mask is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{Mask})$   
 7241         must be from one of the pre-defined types of Table 3.2.
- 7242     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$  of the input matrix.
- 7243     3. If accum is not GrB\_NULL, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 7244         of the accumulation operator and  $\mathbf{D}(A)$  of the input matrix must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
 7245         of the accumulation operator.

7246 Two domains are compatible with each other if values from one domain can be cast to values in  
 7247 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 7248 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 7249 any compatibility rule above is violated, execution of GrB\_transpose ends and the domain mismatch  
 7250 error listed above is returned.

7251 From the argument matrices, the internal matrices and mask used in the computation are formed  
 7252 ( $\leftarrow$  denotes copy):

- 7253     1. Matrix  $\tilde{C} \leftarrow C$ .
- 7254     2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument Mask as follows:

- 7255 (a) If  $\text{Mask} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
7256  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
- 7257 (b) If  $\text{Mask} \neq \text{GrB\_NULL}$ ,
- 7258 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
7259  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
- 7260 ii. Otherwise,  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
7261  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
- 7262 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{M}} \leftarrow \neg \widetilde{\mathbf{M}}$ .
- 7263 3. Matrix  $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

7264 The internal matrices and masks are checked for dimension compatibility. The following conditions  
7265 must hold:

- 7266 1.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$ .
- 7267 2.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$ .
- 7268 3.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{A}})$ .
- 7269 4.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$ .

7270 If any compatibility rule above is violated, execution of `GrB_transpose` ends and the dimension  
7271 mismatch error listed above is returned.

7272 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
7273 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

7274 We are now ready to carry out the matrix transposition and any additional associated operations.  
7275 We describe this in terms of two intermediate matrices:

- 7276 •  $\widetilde{\mathbf{T}}$ : The matrix holding the transpose of  $\widetilde{\mathbf{A}}$ .
- 7277 •  $\widetilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

7278 The intermediate matrix

$$7279 \quad \widetilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \{(j, i, A_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{A}})\} \rangle$$

7280 is created.

7281 The intermediate matrix  $\widetilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 7282 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$ .
- 7283 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{Z}}$  is defined as

$$7284 \quad \widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\} \rangle.$$

7285 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
7286 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned}
7287 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\
7288 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
7289 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
7290 \quad & \\
7291 \quad &
\end{aligned}$$

7292 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

7293 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
7294 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
7295 mask which acts as a “write mask”.

- 7296 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
7297 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$7298 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7299 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
7300 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
7301 mask are unchanged:

$$7302 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7303 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
7304 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
7305 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
7306 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
7307 sequence.

#### 7308 4.3.12 kronecker: Kronecker product of two matrices

7309 Computes the Kronecker product of two matrices. The result is a matrix.

#### 7310 C Syntax

```

7311      GrB_Info GrB_kronecker(GrB_Matrix      C,
7312                           const GrB_Matrix  Mask,
7313                           const GrB_BinaryOp accum,
7314                           const GrB_Semiring op,
7315                           const GrB_Matrix  A,
7316                           const GrB_Matrix  B,
7317                           const GrB_Descriptor desc);
7318
```

```

7319     GrB_Info GrB_kronecker(GrB_Matrix      C,
7320                           const GrB_Matrix  Mask,
7321                           const GrB_BinaryOp accum,
7322                           const GrB_Monoid   op,
7323                           const GrB_Matrix  A,
7324                           const GrB_Matrix  B,
7325                           const GrB_Descriptor desc);
7326
7327     GrB_Info GrB_kronecker(GrB_Matrix      C,
7328                           const GrB_Matrix  Mask,
7329                           const GrB_BinaryOp accum,
7330                           const GrB_BinaryOp op,
7331                           const GrB_Matrix  A,
7332                           const GrB_Matrix  B,
7333                           const GrB_Descriptor desc);

```

## 7334 Parameters

7335 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
7336 that may be accumulated with the result of the Kronecker product. On output,  
7337 the matrix holds the results of the operation.

7338 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
7339 stored into the output matrix C. The mask dimensions must match those of the  
7340 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
7341 of the Mask matrix must be of type bool or any of the predefined “built-in” types  
7342 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
7343 dimensions of C), GrB\_NULL should be specified.

7344 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
7345 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
7346 specified.

7347 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”  
7348 operation. Depending on which type is passed, the following defines the binary  
7349 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

7350 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

7351 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-  
7352 nored.

7353 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  
7354 is ignored.

7355 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
7356 product.

7357 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
7358 product.

7359 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
7360 should be specified. Non-default field/value pairs are listed as follows:  
7361

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.

## 7363 Return Values

7364 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
7365 blocking mode, this indicates that the compatibility tests on di-  
7366 mensions and domains for the input arguments passed successfully.  
7367 Either way, output matrix C is ready to be used in the next method  
7368 of the sequence.

7369 GrB\_PANIC Unknown internal error.

7370 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
7371 GraphBLAS objects (input or output) is in an invalid state caused  
7372 by a previous execution error. Call GrB\_error() to access any error  
7373 messages generated by the implementation.

7374 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

7375 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
7376 a call to new (or Matrix\_dup for matrix parameters).

7377 GrB\_DIMENSION\_MISMATCH Mask and/or matrix dimensions are incompatible.

7378 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
7379 corresponding domains of the binary operator (op) or accumulation  
7380 operator, or the mask's domain is not compatible with bool (in the  
7381 case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

## 7382 Description

7383 GrB\_kronecker computes the Kronecker product  $C = A \otimes B$  or, if an optional binary accumulation  
7384 operator ( $\odot$ ) is provided,  $C = C \odot (A \otimes B)$  (where matrices A and B can be optionally transposed).

7385 The Kronecker product is defined as follows:

7386

$$7387 \quad C = A \otimes B = \begin{bmatrix} A_{0,0} \otimes B & A_{0,1} \otimes B & \dots & A_{0,n_A-1} \otimes B \\ A_{1,0} \otimes B & A_{1,1} \otimes B & \dots & A_{1,n_A-1} \otimes B \\ \vdots & \vdots & \ddots & \vdots \\ A_{m_A-1,0} \otimes B & A_{m_A-1,1} \otimes B & \dots & A_{m_A-1,n_A-1} \otimes B \end{bmatrix}$$

7388 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  
7389 Kronecker product are defined as

$$7390 \quad C(i_A m_B + i_B, j_A n_B + j_B) = A_{i_A, j_A} \otimes B_{i_B, j_B},$$

7391 where  $\otimes$  is the multiplicative operator specified by the **op** parameter.

7392 Logically, this operation occurs in three steps:

7393 **Setup** The internal matrices and mask used in the computation are formed and their domains  
7394 and dimensions are tested for compatibility.

7395 **Compute** The indicated computations are carried out.

7396 **Output** The result is written into the output matrix, possibly under control of a mask.

7397 Up to four argument matrices are used in the **GrB\_kronecker** operation:

- 7398 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7399 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)
- 7400 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 7401 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

7402 The argument matrices, the "product" operator (**op**), and the accumulation operator (if provided)  
7403 are tested for domain compatibility as follows:

- 7404 1. If **Mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\text{Mask})$   
7405 must be from one of the pre-defined types of Table 3.2.
- 7406 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$ .
- 7407 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$ .
- 7408 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 7409 5. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
7410 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of **op** must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
7411 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.



7445 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}) \times \mathbf{nrows}(\tilde{\mathbf{B}}), \mathbf{ncols}(\tilde{\mathbf{A}}) \times \mathbf{ncols}(\tilde{\mathbf{B}}), \{(i, j, T_{ij}) \text{ where } i =$   
7446  $i_A \cdot m_B + i_B, j = j_A \cdot n_B + j_B, \forall (i_A, j_A) = \mathbf{ind}(\tilde{\mathbf{A}}), (i_B, j_B) = \mathbf{ind}(\tilde{\mathbf{B}})\}$  is created. The value of  
7447 each of its elements is computed by

$$7448 \quad T_{i_A \cdot m_B + i_B, j_A \cdot n_B + j_B} = \tilde{\mathbf{A}}(i_A, j_A) \otimes \tilde{\mathbf{B}}(i_B, j_B),$$

7449 where  $\otimes$  is the multiplicative operator specified by the `op` parameter.

7450 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 7451 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 7452 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$7453 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

7454 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
7455 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$7456 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$7457 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$7459 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

7461 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

7462 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
7463 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
7464 mask which acts as a “write mask”.

- 7465 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
7466 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$7467 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7468 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
7469 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
7470 mask are unchanged:

$$7471 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7472 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
7473 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
7474 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
7475 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
7476 sequence. s



## Chapter 5

# Nonpolymorphic interface[Scott: NEW CONTENT]

Each polymorphic GraphBLAS method (those with multiple parameter signatures under the same name) has a corresponding set of long-name forms that are specific to each parameter signature. That is show in Tables 5.1 through 5.11.

Table 5.1: Long-name, nonpolymorphic form of GraphBLAS methods.

Polymorphic signature	Nonpolymorphic signature
GrB_Monoid_new(GrB_Monoid*,...,bool)	GrB_Monoid_new_BOOL(GrB_Monoid*,GrB_BinaryOp,bool)
GrB_Monoid_new(GrB_Monoid*,...,int8_t)	GrB_Monoid_new_INT8(GrB_Monoid*,GrB_BinaryOp,int8_t)
GrB_Monoid_new(GrB_Monoid*,...,uint8_t)	GrB_Monoid_new_UINT8(GrB_Monoid*,GrB_BinaryOp,uint8_t)
GrB_Monoid_new(GrB_Monoid*,...,int16_t)	GrB_Monoid_new_INT16(GrB_Monoid*,GrB_BinaryOp,int16_t)
GrB_Monoid_new(GrB_Monoid*,...,uint16_t)	GrB_Monoid_new_UINT16(GrB_Monoid*,GrB_BinaryOp,uint16_t)
GrB_Monoid_new(GrB_Monoid*,...,int32_t)	GrB_Monoid_new_INT32(GrB_Monoid*,GrB_BinaryOp,int32_t)
GrB_Monoid_new(GrB_Monoid*,...,uint32_t)	GrB_Monoid_new_UINT32(GrB_Monoid*,GrB_BinaryOp,uint32_t)
GrB_Monoid_new(GrB_Monoid*,...,int64_t)	GrB_Monoid_new_INT64(GrB_Monoid*,GrB_BinaryOp,int64_t)
GrB_Monoid_new(GrB_Monoid*,...,uint64_t)	GrB_Monoid_new_UINT64(GrB_Monoid*,GrB_BinaryOp,uint64_t)
GrB_Monoid_new(GrB_Monoid*,...,float)	GrB_Monoid_new_FP32(GrB_Monoid*,GrB_BinaryOp,float)
GrB_Monoid_new(GrB_Monoid*,...,double)	GrB_Monoid_new_FP64(GrB_Monoid*,GrB_BinaryOp,double)
GrB_Monoid_new(GrB_Monoid*,...,other)	GrB_Monoid_new_UDT(GrB_Monoid*,GrB_BinaryOp,void*)

Table 5.2: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Scalar_setElement(..., bool,...)	GrB_Scalar_setElement_BOOL(..., bool,...)
GrB_Scalar_setElement(..., int8_t,...)	GrB_Scalar_setElement_INT8(..., int8_t,...)
GrB_Scalar_setElement(..., uint8_t,...)	GrB_Scalar_setElement_UINT8(..., uint8_t,...)
GrB_Scalar_setElement(..., int16_t,...)	GrB_Scalar_setElement_INT16(..., int16_t,...)
GrB_Scalar_setElement(..., uint16_t,...)	GrB_Scalar_setElement_UINT16(..., uint16_t,...)
GrB_Scalar_setElement(..., int32_t,...)	GrB_Scalar_setElement_INT32(..., int32_t,...)
GrB_Scalar_setElement(..., uint32_t,...)	GrB_Scalar_setElement_UINT32(..., uint32_t,...)
GrB_Scalar_setElement(..., int64_t,...)	GrB_Scalar_setElement_INT64(..., int64_t,...)
GrB_Scalar_setElement(..., uint64_t,...)	GrB_Scalar_setElement_UINT64(..., uint64_t,...)
GrB_Scalar_setElement(..., float,...)	GrB_Scalar_setElement_FP32(..., float,...)
GrB_Scalar_setElement(..., double,...)	GrB_Scalar_setElement_FP64(..., double,...)
GrB_Scalar_setElement(..., <i>other</i> ,...)	GrB_Scalar_setElement_UDT(..., const void*,...)
GrB_Scalar_extractElement(bool*,...)	GrB_Scalar_extractElement_BOOL(bool*,...)
GrB_Scalar_extractElement(int8_t*,...)	GrB_Scalar_extractElement_INT8(int8_t*,...)
GrB_Scalar_extractElement(uint8_t*,...)	GrB_Scalar_extractElement_UINT8(uint8_t*,...)
GrB_Scalar_extractElement(int16_t*,...)	GrB_Scalar_extractElement_INT16(int16_t*,...)
GrB_Scalar_extractElement(uint16_t*,...)	GrB_Scalar_extractElement_UINT16(uint16_t*,...)
GrB_Scalar_extractElement(int32_t*,...)	GrB_Scalar_extractElement_INT32(int32_t*,...)
GrB_Scalar_extractElement(uint32_t*,...)	GrB_Scalar_extractElement_UINT32(uint32_t*,...)
GrB_Scalar_extractElement(int64_t*,...)	GrB_Scalar_extractElement_INT64(int64_t*,...)
GrB_Scalar_extractElement(uint64_t*,...)	GrB_Scalar_extractElement_UINT64(uint64_t*,...)
GrB_Scalar_extractElement(float*,...)	GrB_Scalar_extractElement_FP32(float*,...)
GrB_Scalar_extractElement(double*,...)	GrB_Scalar_extractElement_FP64(double*,...)
GrB_Scalar_extractElement( <i>other</i> *,...)	GrB_Scalar_extractElement_UDT(void*,...)

Table 5.3: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Vector_build(...,const bool*,...)	GrB_Vector_build_BOOL(...,const bool*,...)
GrB_Vector_build(...,const int8_t*,...)	GrB_Vector_build_INT8(...,const int8_t*,...)
GrB_Vector_build(...,const uint8_t*,...)	GrB_Vector_build_UINT8(...,const uint8_t*,...)
GrB_Vector_build(...,const int16_t*,...)	GrB_Vector_build_INT16(...,const int16_t*,...)
GrB_Vector_build(...,const uint16_t*,...)	GrB_Vector_build_UINT16(...,const uint16_t*,...)
GrB_Vector_build(...,const int32_t*,...)	GrB_Vector_build_INT32(...,const int32_t*,...)
GrB_Vector_build(...,const uint32_t*,...)	GrB_Vector_build_UINT32(...,const uint32_t*,...)
GrB_Vector_build(...,const int64_t*,...)	GrB_Vector_build_INT64(...,const int64_t*,...)
GrB_Vector_build(...,const uint64_t*,...)	GrB_Vector_build_UINT64(...,const uint64_t*,...)
GrB_Vector_build(...,const float*,...)	GrB_Vector_build_FP32(...,const float*,...)
GrB_Vector_build(...,const double*,...)	GrB_Vector_build_FP64(...,const double*,...)
GrB_Vector_build(...,const <i>other</i> *,...)	GrB_Vector_build_UDT(...,const void*,...)
GrB_Vector_setElement(...,GrB_Scalar,...)	GrB_Vector_setElement_Scalar(...,const GrB_Scalar,...)
GrB_Vector_setElement(...,bool,...)	GrB_Vector_setElement_BOOL(..., bool,...)
GrB_Vector_setElement(...,int8_t,...)	GrB_Vector_setElement_INT8(..., int8_t,...)
GrB_Vector_setElement(...,uint8_t,...)	GrB_Vector_setElement_UINT8(..., uint8_t,...)
GrB_Vector_setElement(...,int16_t,...)	GrB_Vector_setElement_INT16(..., int16_t,...)
GrB_Vector_setElement(...,uint16_t,...)	GrB_Vector_setElement_UINT16(..., uint16_t,...)
GrB_Vector_setElement(...,int32_t,...)	GrB_Vector_setElement_INT32(..., int32_t,...)
GrB_Vector_setElement(...,uint32_t,...)	GrB_Vector_setElement_UINT32(..., uint32_t,...)
GrB_Vector_setElement(...,int64_t,...)	GrB_Vector_setElement_INT64(..., int64_t,...)
GrB_Vector_setElement(...,uint64_t,...)	GrB_Vector_setElement_UINT64(..., uint64_t,...)
GrB_Vector_setElement(...,float,...)	GrB_Vector_setElement_FP32(..., float,...)
GrB_Vector_setElement(...,double,...)	GrB_Vector_setElement_FP64(..., double,...)
GrB_Vector_setElement(..., <i>other</i> ,...)	GrB_Vector_setElement_UDT(...,const void*,...)
GrB_Vector_extractElement(GrB_Scalar,...)	GrB_Vector_extractElement_Scalar(GrB_Scalar,...)
GrB_Vector_extractElement(bool*,...)	GrB_Vector_extractElement_BOOL(bool*,...)
GrB_Vector_extractElement(int8_t*,...)	GrB_Vector_extractElement_INT8(int8_t*,...)
GrB_Vector_extractElement(uint8_t*,...)	GrB_Vector_extractElement_UINT8(uint8_t*,...)
GrB_Vector_extractElement(int16_t*,...)	GrB_Vector_extractElement_INT16(int16_t*,...)
GrB_Vector_extractElement(uint16_t*,...)	GrB_Vector_extractElement_UINT16(uint16_t*,...)
GrB_Vector_extractElement(int32_t*,...)	GrB_Vector_extractElement_INT32(int32_t*,...)
GrB_Vector_extractElement(uint32_t*,...)	GrB_Vector_extractElement_UINT32(uint32_t*,...)
GrB_Vector_extractElement(int64_t*,...)	GrB_Vector_extractElement_INT64(int64_t*,...)
GrB_Vector_extractElement(uint64_t*,...)	GrB_Vector_extractElement_UINT64(uint64_t*,...)
GrB_Vector_extractElement(float*,...)	GrB_Vector_extractElement_FP32(float*,...)
GrB_Vector_extractElement(double*,...)	GrB_Vector_extractElement_FP64(double*,...)
GrB_Vector_extractElement( <i>other</i> *,...)	GrB_Vector_extractElement_UDT(void*,...)
GrB_Vector_extractTuples(...,bool*,...)	GrB_Vector_extractTuples_BOOL(..., bool*,...)
GrB_Vector_extractTuples(...,int8_t*,...)	GrB_Vector_extractTuples_INT8(..., int8_t*,...)
GrB_Vector_extractTuples(...,uint8_t*,...)	GrB_Vector_extractTuples_UINT8(..., uint8_t*,...)
GrB_Vector_extractTuples(...,int16_t*,...)	GrB_Vector_extractTuples_INT16(..., int16_t*,...)
GrB_Vector_extractTuples(...,uint16_t*,...)	GrB_Vector_extractTuples_UINT16(..., uint16_t*,...)
GrB_Vector_extractTuples(...,int32_t*,...)	GrB_Vector_extractTuples_INT32(..., int32_t*,...)
GrB_Vector_extractTuples(...,uint32_t*,...)	GrB_Vector_extractTuples_UINT32(..., uint32_t*,...)
GrB_Vector_extractTuples(...,int64_t*,...)	GrB_Vector_extractTuples_INT64(..., int64_t*,...)
GrB_Vector_extractTuples(...,uint64_t*,...)	GrB_Vector_extractTuples_UINT64(..., uint64_t*,...)
GrB_Vector_extractTuples(...,float*,...)	GrB_Vector_extractTuples_FP32(..., float*,...)
GrB_Vector_extractTuples(...,double*,...)	GrB_Vector_extractTuples_FP64(..., double*,...)
GrB_Vector_extractTuples(..., <i>other</i> *,...)	GrB_Vector_extractTuples_UDT(..., void*,...)

Table 5.4: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Matrix_build(...,const bool*,...)	GrB_Matrix_build_BOOL(...,const bool*,...)
GrB_Matrix_build(...,const int8_t*,...)	GrB_Matrix_build_INT8(...,const int8_t*,...)
GrB_Matrix_build(...,const uint8_t*,...)	GrB_Matrix_build_UINT8(...,const uint8_t*,...)
GrB_Matrix_build(...,const int16_t*,...)	GrB_Matrix_build_INT16(...,const int16_t*,...)
GrB_Matrix_build(...,const uint16_t*,...)	GrB_Matrix_build_UINT16(...,const uint16_t*,...)
GrB_Matrix_build(...,const int32_t*,...)	GrB_Matrix_build_INT32(...,const int32_t*,...)
GrB_Matrix_build(...,const uint32_t*,...)	GrB_Matrix_build_UINT32(...,const uint32_t*,...)
GrB_Matrix_build(...,const int64_t*,...)	GrB_Matrix_build_INT64(...,const int64_t*,...)
GrB_Matrix_build(...,const uint64_t*,...)	GrB_Matrix_build_UINT64(...,const uint64_t*,...)
GrB_Matrix_build(...,const float*,...)	GrB_Matrix_build_FP32(...,const float*,...)
GrB_Matrix_build(...,const double*,...)	GrB_Matrix_build_FP64(...,const double*,...)
GrB_Matrix_build(...,const <i>other</i> *,...)	GrB_Matrix_build_UDT(...,const void*,...)
GrB_Matrix_setElement(...,GrB_Scalar,...)	GrB_Matrix_setElement_Scalar(...,const GrB_Scalar,...)
GrB_Matrix_setElement(...,bool,...)	GrB_Matrix_setElement_BOOL(..., bool,...)
GrB_Matrix_setElement(...,int8_t,...)	GrB_Matrix_setElement_INT8(..., int8_t,...)
GrB_Matrix_setElement(...,uint8_t,...)	GrB_Matrix_setElement_UINT8(..., uint8_t,...)
GrB_Matrix_setElement(...,int16_t,...)	GrB_Matrix_setElement_INT16(..., int16_t,...)
GrB_Matrix_setElement(...,uint16_t,...)	GrB_Matrix_setElement_UINT16(..., uint16_t,...)
GrB_Matrix_setElement(...,int32_t,...)	GrB_Matrix_setElement_INT32(..., int32_t,...)
GrB_Matrix_setElement(...,uint32_t,...)	GrB_Matrix_setElement_UINT32(..., uint32_t,...)
GrB_Matrix_setElement(...,int64_t,...)	GrB_Matrix_setElement_INT64(..., int64_t,...)
GrB_Matrix_setElement(...,uint64_t,...)	GrB_Matrix_setElement_UINT64(..., uint64_t,...)
GrB_Matrix_setElement(...,float,...)	GrB_Matrix_setElement_FP32(..., float,...)
GrB_Matrix_setElement(...,double,...)	GrB_Matrix_setElement_FP64(..., double,...)
GrB_Matrix_setElement(..., <i>other</i> ,...)	GrB_Matrix_setElement_UDT(...,const void*,...)
GrB_Matrix_extractElement(GrB_Scalar,...)	GrB_Matrix_extractElement_Scalar(GrB_Scalar,...)
GrB_Matrix_extractElement(bool*,...)	GrB_Matrix_extractElement_BOOL(bool*,...)
GrB_Matrix_extractElement(int8_t*,...)	GrB_Matrix_extractElement_INT8(int8_t*,...)
GrB_Matrix_extractElement(uint8_t*,...)	GrB_Matrix_extractElement_UINT8(uint8_t*,...)
GrB_Matrix_extractElement(int16_t*,...)	GrB_Matrix_extractElement_INT16(int16_t*,...)
GrB_Matrix_extractElement(uint16_t*,...)	GrB_Matrix_extractElement_UINT16(uint16_t*,...)
GrB_Matrix_extractElement(int32_t*,...)	GrB_Matrix_extractElement_INT32(int32_t*,...)
GrB_Matrix_extractElement(uint32_t*,...)	GrB_Matrix_extractElement_UINT32(uint32_t*,...)
GrB_Matrix_extractElement(int64_t*,...)	GrB_Matrix_extractElement_INT64(int64_t*,...)
GrB_Matrix_extractElement(uint64_t*,...)	GrB_Matrix_extractElement_UINT64(uint64_t*,...)
GrB_Matrix_extractElement(float*,...)	GrB_Matrix_extractElement_FP32(float*,...)
GrB_Matrix_extractElement(double*,...)	GrB_Matrix_extractElement_FP64(double*,...)
GrB_Matrix_extractElement( <i>other</i> ,...)	GrB_Matrix_extractElement_UDT(void*,...)
GrB_Matrix_extractTuples(..., bool*,...)	GrB_Matrix_extractTuples_BOOL(..., bool*,...)
GrB_Matrix_extractTuples(..., int8_t*,...)	GrB_Matrix_extractTuples_INT8(..., int8_t*,...)
GrB_Matrix_extractTuples(..., uint8_t*,...)	GrB_Matrix_extractTuples_UINT8(..., uint8_t*,...)
GrB_Matrix_extractTuples(..., int16_t*,...)	GrB_Matrix_extractTuples_INT16(..., int16_t*,...)
GrB_Matrix_extractTuples(..., uint16_t*,...)	GrB_Matrix_extractTuples_UINT16(..., uint16_t*,...)
GrB_Matrix_extractTuples(..., int32_t*,...)	GrB_Matrix_extractTuples_INT32(..., int32_t*,...)
GrB_Matrix_extractTuples(..., uint32_t*,...)	GrB_Matrix_extractTuples_UINT32(..., uint32_t*,...)
GrB_Matrix_extractTuples(..., int64_t*,...)	GrB_Matrix_extractTuples_INT64(..., int64_t*,...)
GrB_Matrix_extractTuples(..., uint64_t*,...)	GrB_Matrix_extractTuples_UINT64(..., uint64_t*,...)
GrB_Matrix_extractTuples(..., float*,...)	GrB_Matrix_extractTuples_FP32(..., float*,...)
GrB_Matrix_extractTuples(..., double*,...)	GrB_Matrix_extractTuples_FP64(..., double*,...)
GrB_Matrix_extractTuples(..., <i>other</i> *,...)	GrB_Matrix_extractTuples_UDT(..., void*,...)

Table 5.5: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Matrix_import(...,const bool*,...)	GrB_Matrix_import_BOOL(...,const bool*,...)
GrB_Matrix_import(...,const int8_t*,...)	GrB_Matrix_import_INT8(...,const int8_t*,...)
GrB_Matrix_import(...,const uint8_t*,...)	GrB_Matrix_import_UINT8(...,const uint8_t*,...)
GrB_Matrix_import(...,const int16_t*,...)	GrB_Matrix_import_INT16(...,const int16_t*,...)
GrB_Matrix_import(...,const uint16_t*,...)	GrB_Matrix_import_UINT16(...,const uint16_t*,...)
GrB_Matrix_import(...,const int32_t*,...)	GrB_Matrix_import_INT32(...,const int32_t*,...)
GrB_Matrix_import(...,const uint32_t*,...)	GrB_Matrix_import_UINT32(...,const uint32_t*,...)
GrB_Matrix_import(...,const int64_t*,...)	GrB_Matrix_import_INT64(...,const int64_t*,...)
GrB_Matrix_import(...,const uint64_t*,...)	GrB_Matrix_import_UINT64(...,const uint64_t*,...)
GrB_Matrix_import(...,const float*,...)	GrB_Matrix_import_FP32(...,const float*,...)
GrB_Matrix_import(...,const double*,...)	GrB_Matrix_import_FP64(...,const double*,...)
GrB_Matrix_import(...,const other,...)	GrB_Matrix_import_UDT(...,const void*,...)
GrB_Matrix_export(...,bool*,...)	GrB_Matrix_export_BOOL(...,bool*,...)
GrB_Matrix_export(...,int8_t*,...)	GrB_Matrix_export_INT8(...,int8_t*,...)
GrB_Matrix_export(...,uint8_t*,...)	GrB_Matrix_export_UINT8(...,uint8_t*,...)
GrB_Matrix_export(...,int16_t*,...)	GrB_Matrix_export_INT16(...,int16_t*,...)
GrB_Matrix_export(...,uint16_t*,...)	GrB_Matrix_export_UINT16(...,uint16_t*,...)
GrB_Matrix_export(...,int32_t*,...)	GrB_Matrix_export_INT32(...,int32_t*,...)
GrB_Matrix_export(...,uint32_t*,...)	GrB_Matrix_export_UINT32(...,uint32_t*,...)
GrB_Matrix_export(...,int64_t*,...)	GrB_Matrix_export_INT64(...,int64_t*,...)
GrB_Matrix_export(...,uint64_t*,...)	GrB_Matrix_export_UINT64(...,uint64_t*,...)
GrB_Matrix_export(...,float*,...)	GrB_Matrix_export_FP32(...,float*,...)
GrB_Matrix_export(...,double*,...)	GrB_Matrix_export_FP64(...,double*,...)
GrB_Matrix_export(...,other,...)	GrB_Matrix_export_UDT(...,void*,...)
GrB_free(GrB_Type*)	GrB_Type_free(GrB_Type*)
GrB_free(GrB_UnaryOp*)	GrB_UnaryOp_free(GrB_UnaryOp*)
GrB_free(GrB_IndexUnaryOp*)	GrB_IndexUnaryOp_free(GrB_IndexUnaryOp*)
GrB_free(GrB_BinaryOp*)	GrB_BinaryOp_free(GrB_BinaryOp*)
GrB_free(GrB_Monoid*)	GrB_Monoid_free(GrB_Monoid*)
GrB_free(GrB_Semiring*)	GrB_Semiring_free(GrB_Semiring*)
GrB_free(GrB_Scalar*)	GrB_Scalar_free(GrB_Scalar*)
GrB_free(GrB_Vector*)	GrB_Vector_free(GrB_Vector*)
GrB_free(GrB_Matrix*)	GrB_Matrix_free(GrB_Matrix*)
GrB_free(GrB_Descriptor*)	GrB_Descriptor_free(GrB_Descriptor*)
GrB_wait(GrB_Type, GrB_WaitMode)	GrB_Type_wait(GrB_Type, GrB_WaitMode)
GrB_wait(GrB_UnaryOp, GrB_WaitMode)	GrB_UnaryOp_wait(GrB_UnaryOp, GrB_WaitMode)
GrB_wait(GrB_IndexUnaryOp, GrB_WaitMode)	GrB_IndexUnaryOp_wait(GrB_IndexUnaryOp, GrB_WaitMode)
GrB_wait(GrB_BinaryOp, GrB_WaitMode)	GrB_BinaryOp_wait(GrB_BinaryOp, GrB_WaitMode)
GrB_wait(GrB_Monoid, GrB_WaitMode)	GrB_Monoid_wait(GrB_Monoid, GrB_WaitMode)
GrB_wait(GrB_Semiring, GrB_WaitMode)	GrB_Semiring_wait(GrB_Semiring, GrB_WaitMode)
GrB_wait(GrB_Scalar, GrB_WaitMode)	GrB_Scalar_wait(GrB_Scalar, GrB_WaitMode)
GrB_wait(GrB_Vector, GrB_WaitMode)	GrB_Vector_wait(GrB_Vector, GrB_WaitMode)
GrB_wait(GrB_Matrix, GrB_WaitMode)	GrB_Matrix_wait(GrB_Matrix, GrB_WaitMode)
GrB_wait(GrB_Descriptor, GrB_WaitMode)	GrB_Descriptor_wait(GrB_Descriptor, GrB_WaitMode)
GrB_error(const char**, const GrB_Type)	GrB_Type_error(const char**, const GrB_Type)
GrB_error(const char**, const GrB_UnaryOp)	GrB_UnaryOp_error(const char**, const GrB_UnaryOp)
GrB_error(const char**, const GrB_IndexUnaryOp)	GrB_IndexUnaryOp_error(const char**, const GrB_IndexUnaryOp)
GrB_error(const char**, const GrB_BinaryOp)	GrB_BinaryOp_error(const char**, const GrB_BinaryOp)
GrB_error(const char**, const GrB_Monoid)	GrB_Monoid_error(const char**, const GrB_Monoid)
GrB_error(const char**, const GrB_Semiring)	GrB_Semiring_error(const char**, const GrB_Semiring)
GrB_error(const char**, const GrB_Scalar)	GrB_Scalar_error(const char**, const GrB_Scalar)
GrB_error(const char**, const GrB_Vector)	GrB_Vector_error(const char**, const GrB_Vector)
GrB_error(const char**, const GrB_Matrix)	GrB_Matrix_error(const char**, const GrB_Matrix)
GrB_error(const char**, const GrB_Descriptor)	GrB_Descriptor_error(const char**, const GrB_Descriptor)

Table 5.6: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_eWiseMult(GrB_Vector,...,GrB_Semiring,...)	GrB_Vector_eWiseMult_Semiring(GrB_Vector,...,GrB_Semiring,...)
GrB_eWiseMult(GrB_Vector,...,GrB_Monoid,...)	GrB_Vector_eWiseMult_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_eWiseMult(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Vector_eWiseMult_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_eWiseMult(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_eWiseMult_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_eWiseMult(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_eWiseMult_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_eWiseMult(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_eWiseMult_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)
GrB_eWiseAdd(GrB_Vector,...,GrB_Semiring,...)	GrB_Vector_eWiseAdd_Semiring(GrB_Vector,...,GrB_Semiring,...)
GrB_eWiseAdd(GrB_Vector,...,GrB_Monoid,...)	GrB_Vector_eWiseAdd_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_eWiseAdd(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Vector_eWiseAdd_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_eWiseAdd(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_eWiseAdd_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_eWiseAdd(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_eWiseAdd_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_eWiseAdd(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_eWiseAdd_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)
GrB_extract(GrB_Vector,...,GrB_Vector,...)	GrB_Vector_extract(GrB_Vector,...,GrB_Vector,...)
GrB_extract(GrB_Matrix,...,GrB_Matrix,...)	GrB_Matrix_extract(GrB_Matrix,...,GrB_Matrix,...)
GrB_extract(GrB_Vector,...,GrB_Matrix,...)	GrB_Col_extract(GrB_Vector,...,GrB_Matrix,...)
GrB_assign(GrB_Vector,...,GrB_Vector,...)	GrB_Vector_assign(GrB_Vector,...,GrB_Vector,...)
GrB_assign(GrB_Matrix,...,GrB_Matrix,...)	GrB_Matrix_assign(GrB_Matrix,...,GrB_Matrix,...)
GrB_assign(GrB_Matrix,...,GrB_Vector,const GrB_Index*,...)	GrB_Col_assign(GrB_Matrix,...,GrB_Vector,const GrB_Index*,...)
GrB_assign(GrB_Matrix,...,GrB_Vector,GrB_Index,...)	GrB_Row_assign(GrB_Matrix,...,GrB_Vector,GrB_Index,...)
GrB_assign(GrB_Vector,...,GrB_Scalar,...)	GrB_Vector_assign_Scalar(GrB_Vector,...,const GrB_Scalar,...)
GrB_assign(GrB_Vector,...,bool,...)	GrB_Vector_assign_BOOL(GrB_Vector,..., bool,...)
GrB_assign(GrB_Vector,...,int8_t,...)	GrB_Vector_assign_INT8(GrB_Vector,..., int8_t,...)
GrB_assign(GrB_Vector,...,uint8_t,...)	GrB_Vector_assign_UINT8(GrB_Vector,..., uint8_t,...)
GrB_assign(GrB_Vector,...,int16_t,...)	GrB_Vector_assign_INT16(GrB_Vector,..., int16_t,...)
GrB_assign(GrB_Vector,...,uint16_t,...)	GrB_Vector_assign_UINT16(GrB_Vector,..., uint16_t,...)
GrB_assign(GrB_Vector,...,int32_t,...)	GrB_Vector_assign_INT32(GrB_Vector,..., int32_t,...)
GrB_assign(GrB_Vector,...,uint32_t,...)	GrB_Vector_assign_UINT32(GrB_Vector,..., uint32_t,...)
GrB_assign(GrB_Vector,...,int64_t,...)	GrB_Vector_assign_INT64(GrB_Vector,..., int64_t,...)
GrB_assign(GrB_Vector,...,uint64_t,...)	GrB_Vector_assign_UINT64(GrB_Vector,..., uint64_t,...)
GrB_assign(GrB_Vector,...,float,...)	GrB_Vector_assign_FP32(GrB_Vector,..., float,...)
GrB_assign(GrB_Vector,...,double,...)	GrB_Vector_assign_FP64(GrB_Vector,..., double,...)
GrB_assign(GrB_Vector,...,other,...)	GrB_Vector_assign_UDT(GrB_Vector,...,const void*,...)
GrB_assign(GrB_Matrix,...,GrB_Scalar,...)	GrB_Matrix_assign_Scalar(GrB_Matrix,...,const GrB_Scalar,...)
GrB_assign(GrB_Matrix,...,bool,...)	GrB_Matrix_assign_BOOL(GrB_Matrix,..., bool,...)
GrB_assign(GrB_Matrix,...,int8_t,...)	GrB_Matrix_assign_INT8(GrB_Matrix,..., int8_t,...)
GrB_assign(GrB_Matrix,...,uint8_t,...)	GrB_Matrix_assign_UINT8(GrB_Matrix,..., uint8_t,...)
GrB_assign(GrB_Matrix,...,int16_t,...)	GrB_Matrix_assign_INT16(GrB_Matrix,..., int16_t,...)
GrB_assign(GrB_Matrix,...,uint16_t,...)	GrB_Matrix_assign_UINT16(GrB_Matrix,..., uint16_t,...)
GrB_assign(GrB_Matrix,...,int32_t,...)	GrB_Matrix_assign_INT32(GrB_Matrix,..., int32_t,...)
GrB_assign(GrB_Matrix,...,uint32_t,...)	GrB_Matrix_assign_UINT32(GrB_Matrix,..., uint32_t,...)
GrB_assign(GrB_Matrix,...,int64_t,...)	GrB_Matrix_assign_INT64(GrB_Matrix,..., int64_t,...)
GrB_assign(GrB_Matrix,...,uint64_t,...)	GrB_Matrix_assign_UINT64(GrB_Matrix,..., uint64_t,...)
GrB_assign(GrB_Matrix,...,float,...)	GrB_Matrix_assign_FP32(GrB_Matrix,..., float,...)
GrB_assign(GrB_Matrix,...,double,...)	GrB_Matrix_assign_FP64(GrB_Matrix,..., double,...)
GrB_assign(GrB_Matrix,...,other,...)	GrB_Matrix_assign_UDT(GrB_Matrix,...,const void*,...)



Table 5.7: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_apply(GrB_Vector,...,GrB_UnaryOp,GrB_Vector,...)	GrB_Vector_apply(GrB_Vector,...,GrB_UnaryOp,GrB_Vector,...)
GrB_apply(GrB_Matrix,...,GrB_UnaryOp,GrB_Matrix,...)	GrB_Matrix_apply(GrB_Matrix,...,GrB_UnaryOp,GrB_Matrix,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Scalar,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_Scalar(GrB_Vector,...,GrB_BinaryOp,GrB_Scalar,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,bool,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_BOOL(GrB_Vector,...,GrB_BinaryOp,bool,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int8_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT8(GrB_Vector,...,GrB_BinaryOp,int8_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint8_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT8(GrB_Vector,...,GrB_BinaryOp,uint8_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int16_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT16(GrB_Vector,...,GrB_BinaryOp,int16_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint16_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT16(GrB_Vector,...,GrB_BinaryOp,uint16_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int32_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT32(GrB_Vector,...,GrB_BinaryOp,int32_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint32_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT32(GrB_Vector,...,GrB_BinaryOp,uint32_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int64_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT64(GrB_Vector,...,GrB_BinaryOp,int64_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint64_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT64(GrB_Vector,...,GrB_BinaryOp,uint64_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,float,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_FP32(GrB_Vector,...,GrB_BinaryOp,float,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,double,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_FP64(GrB_Vector,...,GrB_BinaryOp,double,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp, <i>other</i> ,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UDT(GrB_Vector,...,GrB_BinaryOp,const void*,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,GrB_Scalar,...)	GrB_Vector_apply_BinaryOp2nd_Scalar(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,GrB_Scalar,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,bool,...)	GrB_Vector_apply_BinaryOp2nd_BOOL(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,bool,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int8_t,...)	GrB_Vector_apply_BinaryOp2nd_INT8(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int8_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint8_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT8(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint8_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int16_t,...)	GrB_Vector_apply_BinaryOp2nd_INT16(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int16_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint16_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT16(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint16_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int32_t,...)	GrB_Vector_apply_BinaryOp2nd_INT32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int32_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint32_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint32_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int64_t,...)	GrB_Vector_apply_BinaryOp2nd_INT64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int64_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint64_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint64_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,float,...)	GrB_Vector_apply_BinaryOp2nd_FP32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,float,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,double,...)	GrB_Vector_apply_BinaryOp2nd_FP64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,double,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector, <i>other</i> ,...)	GrB_Vector_apply_BinaryOp2nd_UDT(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,const void*,...)

Table 5.8: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Scalar,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_Scalar(GrB_Matrix,...,GrB_BinaryOp,GrB_Scalar,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,bool,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_BOOL(GrB_Matrix,...,GrB_BinaryOp,bool,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int8_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT8(GrB_Matrix,...,GrB_BinaryOp,int8_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint8_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT8(GrB_Matrix,...,GrB_BinaryOp,uint8_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int16_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT16(GrB_Matrix,...,GrB_BinaryOp,int16_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint16_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT16(GrB_Matrix,...,GrB_BinaryOp,uint16_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int32_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT32(GrB_Matrix,...,GrB_BinaryOp,int32_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint32_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT32(GrB_Matrix,...,GrB_BinaryOp,uint32_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int64_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT64(GrB_Matrix,...,GrB_BinaryOp,int64_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint64_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT64(GrB_Matrix,...,GrB_BinaryOp,uint64_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,float,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_FP32(GrB_Matrix,...,GrB_BinaryOp,float,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,double,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_FP64(GrB_Matrix,...,GrB_BinaryOp,double,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp, <i>other</i> ,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UDT(GrB_Matrix,...,GrB_BinaryOp,const void*,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,GrB_Scalar,...)	GrB_Matrix_apply_BinaryOp2nd_Scalar(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,GrB_Scalar,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,bool,...)	GrB_Matrix_apply_BinaryOp2nd_BOOL(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,bool,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int8_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT8(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int8_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint8_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT8(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint8_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int16_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT16(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int16_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint16_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT16(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint16_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int32_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int32_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint32_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint32_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int64_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int64_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint64_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint64_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,float,...)	GrB_Matrix_apply_BinaryOp2nd_FP32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,float,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,double,...)	GrB_Matrix_apply_BinaryOp2nd_FP64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,double,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix, <i>other</i> ,...)	GrB_Matrix_apply_BinaryOp2nd_UDT(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,const void*,...)

Table 5.9: Long-name, nonpolymorphic form of GraphBLAS methods (continued).[\[Scott: NEW CONTENT\]](#)

Polymorphic signature	Nonpolymorphic signature
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)	GrB_Vector_apply_IndexOp_Scalar(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)	GrB_Vector_apply_IndexOp_BOOL(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)	GrB_Vector_apply_IndexOp_INT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)	GrB_Vector_apply_IndexOp_UINT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)	GrB_Vector_apply_IndexOp_INT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)	GrB_Vector_apply_IndexOp_UINT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)	GrB_Vector_apply_IndexOp_INT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)	GrB_Vector_apply_IndexOp_UINT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)	GrB_Vector_apply_IndexOp_INT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)	GrB_Vector_apply_IndexOp_UINT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)	GrB_Vector_apply_IndexOp_FP32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)	GrB_Vector_apply_IndexOp_FP64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector, <i>other</i> ,...)	GrB_Vector_apply_IndexOp_UDT(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,const void*,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)	GrB_Matrix_apply_IndexOp_Scalar(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)	GrB_Matrix_apply_IndexOp_BOOL(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)	GrB_Matrix_apply_IndexOp_INT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)	GrB_Matrix_apply_IndexOp_UINT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)	GrB_Matrix_apply_IndexOp_INT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)	GrB_Matrix_apply_IndexOp_UINT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)	GrB_Matrix_apply_IndexOp_INT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)	GrB_Matrix_apply_IndexOp_UINT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)	GrB_Matrix_apply_IndexOp_INT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)	GrB_Matrix_apply_IndexOp_UINT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)	GrB_Matrix_apply_IndexOp_FP32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)	GrB_Matrix_apply_IndexOp_FP64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix, <i>other</i> ,...)	GrB_Matrix_apply_IndexOp_UDT(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,const void*,...)

Table 5.10: Long-name, nonpolymorphic form of GraphBLAS methods (continued).[\[Scott: NEW CONTENT\]](#)

Polymorphic signature	Nonpolymorphic signature
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)</code>	<code>GrB_Vector_select_Scalar(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)</code>	<code>GrB_Vector_select_BOOL(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)</code>	<code>GrB_Vector_select_INT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)</code>	<code>GrB_Vector_select_UINT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)</code>	<code>GrB_Vector_select_INT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)</code>	<code>GrB_Vector_select_UINT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)</code>	<code>GrB_Vector_select_INT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)</code>	<code>GrB_Vector_select_UINT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)</code>	<code>GrB_Vector_select_INT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)</code>	<code>GrB_Vector_select_UINT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)</code>	<code>GrB_Vector_select_FP32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)</code>	<code>GrB_Vector_select_FP64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,other,...)</code>	<code>GrB_Vector_select_UDT(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,const void*,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)</code>	<code>GrB_Matrix_select_Scalar(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)</code>	<code>GrB_Matrix_select_BOOL(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)</code>	<code>GrB_Matrix_select_INT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)</code>	<code>GrB_Matrix_select_UINT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)</code>	<code>GrB_Matrix_select_INT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)</code>	<code>GrB_Matrix_select_UINT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)</code>	<code>GrB_Matrix_select_INT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)</code>	<code>GrB_Matrix_select_UINT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)</code>	<code>GrB_Matrix_select_INT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)</code>	<code>GrB_Matrix_select_UINT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)</code>	<code>GrB_Matrix_select_FP32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)</code>	<code>GrB_Matrix_select_FP64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,other,...)</code>	<code>GrB_Matrix_select_UDT(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,const void*,...)</code>

Table 5.11: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_reduce(GrB_Vector,...,GrB_Monoid,...)	GrB_Matrix_reduce_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_reduce(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Matrix_reduce_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_reduce(GrB_Scalar,...,GrB_Monoid,GrB_Vector,...)	GrB_Vector_reduce_Monoid_Scalar(GrB_Scalar,...,GrB_Vector,...)
GrB_reduce(GrB_Scalar,...,GrB_BinaryOp,GrB_Vector,...)	GrB_Vector_reduce_BinaryOp_Scalar(GrB_Scalar,...,GrB_Vector,...)
GrB_reduce(bool*,...,GrB_Vector,...)	GrB_Vector_reduce_BOOL(bool*,...,GrB_Vector,...)
GrB_reduce(int8_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT8(int8_t*,...,GrB_Vector,...)
GrB_reduce(uint8_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT8(uint8_t*,...,GrB_Vector,...)
GrB_reduce(int16_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT16(int16_t*,...,GrB_Vector,...)
GrB_reduce(uint16_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT16(uint16_t*,...,GrB_Vector,...)
GrB_reduce(int32_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT32(int32_t*,...,GrB_Vector,...)
GrB_reduce(uint32_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT32(uint32_t*,...,GrB_Vector,...)
GrB_reduce(int64_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT64(int64_t*,...,GrB_Vector,...)
GrB_reduce(uint64_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT64(uint64_t*,...,GrB_Vector,...)
GrB_reduce(float*,...,GrB_Vector,...)	GrB_Vector_reduce_FP32(float*,...,GrB_Vector,...)
GrB_reduce(double*,...,GrB_Vector,...)	GrB_Vector_reduce_FP64(double*,...,GrB_Vector,...)
GrB_reduce( <i>other</i> *,...,GrB_Vector,...)	GrB_Vector_reduce_UDT(void*,...,GrB_Vector,...)
GrB_reduce(GrB_Scalar,...,GrB_Monoid,GrB_Matrix,...)	GrB_Matrix_reduce_Monoid_Scalar(GrB_Scalar,...,GrB_Monoid,GrB_Matrix,...)
GrB_reduce(GrB_Scalar,...,GrB_BinaryOp,GrB_Matrix,...)	GrB_Matrix_reduce_BinaryOp_Scalar(GrB_Scalar,...,GrB_BinaryOp,GrB_Matrix,...)
GrB_reduce(bool*,...,GrB_Matrix,...)	GrB_Matrix_reduce_BOOL(bool*,...,GrB_Matrix,...)
GrB_reduce(int8_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT8(int8_t*,...,GrB_Matrix,...)
GrB_reduce(uint8_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT8(uint8_t*,...,GrB_Matrix,...)
GrB_reduce(int16_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT16(int16_t*,...,GrB_Matrix,...)
GrB_reduce(uint16_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT16(uint16_t*,...,GrB_Matrix,...)
GrB_reduce(int32_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT32(int32_t*,...,GrB_Matrix,...)
GrB_reduce(uint32_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT32(uint32_t*,...,GrB_Matrix,...)
GrB_reduce(int64_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT64(int64_t*,...,GrB_Matrix,...)
GrB_reduce(uint64_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT64(uint64_t*,...,GrB_Matrix,...)
GrB_reduce(float*,...,GrB_Matrix,...)	GrB_Matrix_reduce_FP32(float*,...,GrB_Matrix,...)
GrB_reduce(double*,...,GrB_Matrix,...)	GrB_Matrix_reduce_FP64(double*,...,GrB_Matrix,...)
GrB_reduce( <i>other</i> *,...,GrB_Matrix,...)	GrB_Matrix_reduce_UDT(void*,...,GrB_Matrix,...)
GrB_kronecker(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_kronecker_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_kronecker(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_kronecker_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_kronecker(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_kronecker_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)

Table 5.12: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Scalar_get(GrB_Scalar,...,GrB_Scalar)	GrB_Scalar_get_Scalar(GrB_Scalar,...,GrB_Scalar)
GrB_Scalar_get(GrB_Scalar,...,char*)	GrB_Scalar_get_String(GrB_Scalar,...,char*)
GrB_Scalar_get(GrB_Scalar,...,int*)	GrB_Scalar_get_INT32(GrB_Scalar,...,int*)
GrB_Scalar_get(GrB_Scalar,...,void*)	GrB_Scalar_get_VOID(GrB_Scalar,...,void*)
GrB_Vector_get(GrB_Vector,...,GrB_Scalar)	GrB_Vector_get_Scalar(GrB_Vector,...,GrB_Scalar)
GrB_Vector_get(GrB_Vector,...,char*)	GrB_Vector_get_String(GrB_Vector,...,char*)
GrB_Vector_get(GrB_Vector,...,int*)	GrB_Matrix_get_INT32(GrB_Vector,...,int*)
GrB_Vector_get(GrB_Vector,...,void*)	GrB_Vector_get_VOID(GrB_Vector,...,void*)
GrB_Matrix_get(GrB_Matrix,...,GrB_Scalar)	GrB_Matrix_get_Scalar(GrB_Matrix,...,GrB_Scalar)
GrB_Matrix_get(GrB_Matrix,...,char*)	GrB_Matrix_get_String(GrB_Matrix,...,char*)
GrB_Matrix_get(GrB_Matrix,...,int*)	GrB_Matrix_get_INT32(GrB_Matrix,...,int*)
GrB_Matrix_get(GrB_Matrix,...,void*)	GrB_Matrix_get_VOID(GrB_Matrix,...,void*)
GrB_UnaryOp_get(GrB_UnaryOp,...,GrB_Scalar)	GrB_UnaryOp_get_Scalar(GrB_UnaryOp,...,GrB_Scalar)
GrB_UnaryOp_get(GrB_UnaryOp,...,char*)	GrB_UnaryOp_get_String(GrB_UnaryOp,...,char*)
GrB_UnaryOp_get(GrB_UnaryOp,...,int*)	GrB_UnaryOp_get_INT32(GrB_UnaryOp,...,int*)
GrB_UnaryOp_get(GrB_UnaryOp,...,void*)	GrB_UnaryOp_get_VOID(GrB_UnaryOp,...,void*)
GrB_IndexUnaryOp_get(GrB_IndexUnaryOp,...,GrB_Scalar)	GrB_IndexUnaryOp_get_Scalar(GrB_IndexUnaryOp,...,GrB_Scalar)
GrB_IndexUnaryOp_get(GrB_IndexUnaryOp,...,char*)	GrB_IndexUnaryOp_get_String(GrB_IndexUnaryOp,...,char*)
GrB_IndexUnaryOp_get(GrB_IndexUnaryOp,...,int*)	GrB_IndexUnaryOp_get_INT32(GrB_IndexUnaryOp,...,int*)
GrB_IndexUnaryOp_get(GrB_IndexUnaryOp,...,void*)	GrB_IndexUnaryOp_get_VOID(GrB_IndexUnaryOp,...,void*)
GrB_BinaryOp_get(GrB_BinaryOp,...,GrB_Scalar)	GrB_BinaryOp_get_Scalar(GrB_BinaryOp,...,GrB_Scalar)
GrB_BinaryOp_get(GrB_BinaryOp,...,char*)	GrB_BinaryOp_get_String(GrB_BinaryOp,...,char*)
GrB_BinaryOp_get(GrB_BinaryOp,...,int*)	GrB_BinaryOp_get_INT32(GrB_BinaryOp,...,int*)
GrB_BinaryOp_get(GrB_BinaryOp,...,void*)	GrB_BinaryOp_get_VOID(GrB_BinaryOp,...,void*)
GrB_Monoid_get(GrB_Monoid,...,GrB_Scalar)	GrB_Monoid_get_Scalar(GrB_Monoid,...,GrB_Scalar)
GrB_Monoid_get(GrB_Monoid,...,char*)	GrB_Monoid_get_String(GrB_Monoid,...,char*)
GrB_Monoid_get(GrB_Monoid,...,int*)	GrB_Monoid_get_INT32(GrB_Monoid,...,int*)
GrB_Monoid_get(GrB_Monoid,...,void*)	GrB_Monoid_get_VOID(GrB_Monoid,...,void*)
GrB_Semiring_get(GrB_Semiring,...,GrB_Scalar)	GrB_Semiring_get_Scalar(GrB_Semiring,...,GrB_Scalar)
GrB_Semiring_get(GrB_Semiring,...,char*)	GrB_Semiring_get_String(GrB_Semiring,...,char*)
GrB_Semiring_get(GrB_Semiring,...,int*)	GrB_Semiring_get_INT32(GrB_Semiring,...,int*)
GrB_Semiring_get(GrB_Semiring,...,void*)	GrB_Semiring_get_VOID(GrB_Semiring,...,void*)
GrB_Descriptor_get(GrB_Descriptor,...,GrB_Scalar)	GrB_Descriptor_get_Scalar(GrB_Descriptor,...,GrB_Scalar)
GrB_Descriptor_get(GrB_Descriptor,...,char*)	GrB_Descriptor_get_String(GrB_Descriptor,...,char*)
GrB_Descriptor_get(GrB_Descriptor,...,int*)	GrB_Descriptor_get_INT32(GrB_Descriptor,...,int*)
GrB_Descriptor_get(GrB_Descriptor,...,void*)	GrB_Descriptor_get_VOID(GrB_Descriptor,...,void*)
GrB_Type_get(GrB_Type,...,GrB_Scalar)	GrB_Type_get_Scalar(GrB_Type,...,GrB_Scalar)
GrB_Type_get(GrB_Type,...,char*)	GrB_Type_get_String(GrB_Type,...,char*)
GrB_Type_get(GrB_Type,...,int*)	GrB_Type_get_INT32(GrB_Type,...,int*)
GrB_Type_get(GrB_Type,...,void*)	GrB_Type_get_VOID(GrB_Type,...,void*)
GrB_Global_get(...,GrB_Scalar)	GrB_Global_get_Scalar(...,GrB_Scalar)
GrB_Global_get(...,char*)	GrB_Global_get_String(...,char*)
GrB_Global_get(...,int*)	GrB_Global_get_INT32(...,int*)
GrB_Global_get(...,void*)	GrB_Global_get_VOID(...,void*)

Table 5.13: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Scalar_set(GrB_Scalar,...,GrB_Scalar)	GrB_Scalar_set_Scalar(GrB_Scalar,...,GrB_Scalar)
GrB_Scalar_set(GrB_Scalar,...,char*)	GrB_Scalar_set_String(GrB_Scalar,...,char*)
GrB_Scalar_set(GrB_Scalar,...,void*)	GrB_Scalar_set_VOID(GrB_Scalar,...,void*)
GrB_Vector_set(GrB_Vector,...,GrB_Scalar)	GrB_Vector_set_Scalar(GrB_Vector,...,GrB_Scalar)
GrB_Vector_set(GrB_Vector,...,char*)	GrB_Vector_set_String(GrB_Vector,...,char*)
GrB_Vector_set(GrB_Vector,...,void*)	GrB_Vector_set_VOID(GrB_Vector,...,void*)
GrB_Matrix_set(GrB_Matrix,...,GrB_Scalar)	GrB_Matrix_set_Scalar(GrB_Matrix,...,GrB_Scalar)
GrB_Matrix_set(GrB_Matrix,...,char*)	GrB_Matrix_set_String(GrB_Matrix,...,char*)
GrB_Matrix_set(GrB_Matrix,...,void*)	GrB_Matrix_set_VOID(GrB_Matrix,...,void*)
GrB_UnaryOp_set(GrB_UnaryOp,...,GrB_Scalar)	GrB_UnaryOp_set_Scalar(GrB_UnaryOp,...,GrB_Scalar)
GrB_UnaryOp_set(GrB_UnaryOp,...,char*)	GrB_UnaryOp_set_String(GrB_UnaryOp,...,char*)
GrB_UnaryOp_set(GrB_UnaryOp,...,void*)	GrB_UnaryOp_set_VOID(GrB_UnaryOp,...,void*)
GrB_IndexUnaryOp_set(GrB_IndexUnaryOp,...,GrB_Scalar)	GrB_IndexUnaryOp_set_Scalar(GrB_IndexUnaryOp,...,GrB_Scalar)
GrB_IndexUnaryOp_set(GrB_IndexUnaryOp,...,char*)	GrB_IndexUnaryOp_set_String(GrB_IndexUnaryOp,...,char*)
GrB_IndexUnaryOp_set(GrB_IndexUnaryOp,...,void*)	GrB_IndexUnaryOp_set_VOID(GrB_IndexUnaryOp,...,void*)
GrB_BinaryOp_set(GrB_BinaryOp,...,GrB_Scalar)	GrB_BinaryOp_set_Scalar(GrB_BinaryOp,...,GrB_Scalar)
GrB_BinaryOp_set(GrB_BinaryOp,...,char*)	GrB_BinaryOp_set_String(GrB_BinaryOp,...,char*)
GrB_BinaryOp_set(GrB_BinaryOp,...,void*)	GrB_BinaryOp_set_VOID(GrB_BinaryOp,...,void*)
GrB_Monoid_set(GrB_Monoid,...,GrB_Scalar)	GrB_Monoid_set_Scalar(GrB_Monoid,...,GrB_Scalar)
GrB_Monoid_set(GrB_Monoid,...,char*)	GrB_Monoid_set_String(GrB_Monoid,...,char*)
GrB_Monoid_set(GrB_Monoid,...,void*)	GrB_Monoid_set_VOID(GrB_Monoid,...,void*)
GrB_Semiring_set(GrB_Semiring,...,GrB_Scalar)	GrB_Semiring_set_Scalar(GrB_Semiring,...,GrB_Scalar)
GrB_Semiring_set(GrB_Semiring,...,char*)	GrB_Semiring_set_String(GrB_Semiring,...,char*)
GrB_Semiring_set(GrB_Semiring,...,void*)	GrB_Semiring_set_VOID(GrB_Semiring,...,void*)
GrB_Descriptor_set(GrB_Descriptor,...,GrB_Scalar)	GrB_Descriptor_set_Scalar(GrB_Descriptor,...,GrB_Scalar)
GrB_Descriptor_set(GrB_Descriptor,...,char*)	GrB_Descriptor_set_String(GrB_Descriptor,...,char*)
GrB_Descriptor_set(GrB_Descriptor,...,void*)	GrB_Descriptor_set_VOID(GrB_Descriptor,...,void*)
GrB_Type_set(GrB_Type,...,GrB_Scalar)	GrB_Type_set_Scalar(GrB_Type,...,GrB_Scalar)
GrB_Type_set(GrB_Type,...,char*)	GrB_Type_set_String(GrB_Type,...,char*)
GrB_Type_set(GrB_Type,...,void*)	GrB_Type_set_VOID(GrB_Type,...,void*)
GrB_Global_set(...,GrB_Scalar)	GrB_Global_set_Scalar(...,GrB_Scalar)
GrB_Global_set(...,char*)	GrB_Global_set_String(...,char*)
GrB_Global_set(...,void*)	GrB_Global_set_VOID(...,void*)





# 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.

- 7507 • (Issues BB-25, GH-42) Added import and export methods for matrices to/from API specified  
7508 formats. Three formats have been specified: CSC, CSR, COO. Dense row and column formats  
7509 have been deferred.
- 7510 • (Issue BB-75) Added matrix constructor to build a diagonal `GrB_Matrix` from a `GrB_Vector`.
- 7511 • (Issue BB-73) Allow `GrB_NULL` for dup operator in matrix and vector `build` methods. Return  
7512 error if duplicate locations encountered.
- 7513 • (Issue BB-58) Added matrix and vector methods to remove (annihilate) elements.
- 7514 • (Issue BB-17) Added `GrB_ABS_T` (absolute value) unary operator.
- 7515 • (Issue GH-46) Adding `GrB_ONEB_T` binary operator that returns 1 cast to type `T` (not to  
7516 be confused with the proposed unary operator).
- 7517 • (Issue GH-53) Added language about what constitutes a “conformant” implementation. Added  
7518 `GrB_NOT_IMPLEMENTED` return value (API error) for API any combinations of inputs to  
7519 a method that is not supported by the implementation.
- 7520 • Added `GrB_EMPTY_OBJECT` return value (execution error) that is used when an opaque  
7521 object (currently only `GrB_Scalar`) is passed as an input that cannot be empty.
- 7522 • (Issue BB-45) Removed language about annihilators.
- 7523 • (Issue BB-69) Made names/symbols containing underscores searchable in PDF.
- 7524 • Updated a number algorithms in the appendix to use new operations and methods.
- 7525 • Numerous additions (some changes) to the non-polymorphic interface to track changes to the  
7526 specification.
- 7527 • Typographical error in version macros was corrected. They are all caps: `GRB_VERSION` and  
7528 `GRB_SUBVERSION`.
- 7529 • Typographical change to `eWiseAdd` Description to be consistent in order of set intersections.
- 7530 • Typographical errors in `eWiseAdd`: cut-and-paste errors from `eWiseMult`/set intersection  
7531 fixed to read `eWiseAdd`/set union.
- 7532 • Typographical error (`NEQ`  $\rightarrow$  `NE`) in Description of Table 3.8.

7533 Changes in 1.3.0 (Released: 25 September 2019):

- 7534 • (Issue BB-50) Changed definition of completion and added `GrB_wait()` that takes an opaque  
7535 GraphBLAS object as an argument.
- 7536 • (Issue BB-39) Added `GrB_kronecker` operation.
- 7537 • (Issue BB-40) Added variants of the `GrB_apply` operation that take a binary function and a  
7538 scalar.

7539  
7540  
  
7541  
  
7542  
7543  
  
7544  
7545  
  
7546  
  
7547  
7548  
  
7549  
7550  
  
7551  
7552  
  
7553  
  
7554  
7555  
  
7556  
  
7557  
7558  
  
7559  
7560  
  
7561  
  
7562  
7563  
  
7564  
  
7565  
7566  
  
7567  
7568  
  
7569  
  
7570  
  
7571

- (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.

- 7572 • Fixed miscellaneous typographical errors (such as  $\otimes$ ,  $\oplus$ ).
- 7573 Changes in 1.2.0:
- 7574 • Removed "provisional" clause.
- 7575 Changes in 1.1.0:
- 7576 • Removed unnecessary `const` from `nindices`, `nrows`, and `ncols` parameters of both `extract` and
  - 7577 `assign` operations.
  - 7578 • Signature of `GrB_UnaryOp_new` changed: order of input parameters changed.
  - 7579 • Signature of `GrB_BinaryOp_new` changed: order of input parameters changed.
  - 7580 • Signature of `GrB_Monoid_new` changed: removal of domain argument which is now inferred
  - 7581 from the domains of the binary operator provided.
  - 7582 • Signature of `GrB_Vector_extractTuples` and `GrB_Matrix_extractTuples` to add an in/out ar-
  - 7583 gument, `n`, which indicates the size of the output arrays provided (in terms of number of
  - 7584 elements, not number of bytes). Added new execution error, `GrB_INSUFFICIENT_SPACE`
  - 7585 which is returned when the capacities of the output arrays are insufficient to hold all of the
  - 7586 tuples.
  - 7587 • Changed `GrB_Column_assign` to `GrB_Col_assign` for consistency in non-polymorphic inter-
  - 7588 face.
  - 7589 • Added replace flag (`z`) notation to Table 4.1.
  - 7590 • Updated the "Mathematical Description" of the `assign` operation in Table 4.1.
  - 7591 • Added triangle counting example.
  - 7592 • Added subsection headers for `accumulate` and `mask/replace` discussions in the Description
  - 7593 sections of GraphBLAS operations when the respective text was the "standard" text (i.e.,
  - 7594 identical in a majority of the operations).
  - 7595 • Fixed typographical errors.
- 7596 Changes in 1.0.2:
- 7597 • Expanded the definitions of `Vector_build` and `Matrix_build` to conceptually use intermediate
  - 7598 matrices and avoid casting issues in certain implementations.
  - 7599 • Fixed the bug in the `GrB_assign` definition. Elements of the output object are no longer being
  - 7600 erased outside the assigned area.
  - 7601 • Changes non-polymorphic interface:
    - 7602 – Renamed `GrB_Row_extract` to `GrB_Col_extract`.

- 7603           – Renamed GrB\_Vector\_reduce\_BinaryOp to GrB\_Matrix\_reduce\_BinaryOp.
- 7604           – Renamed GrB\_Vector\_reduce\_Monoid to GrB\_Matrix\_reduce\_Monoid.
- 7605       • Fixed the bugs with respect to isolated vertices in the Maximal Independent Set example.
- 7606       • 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.

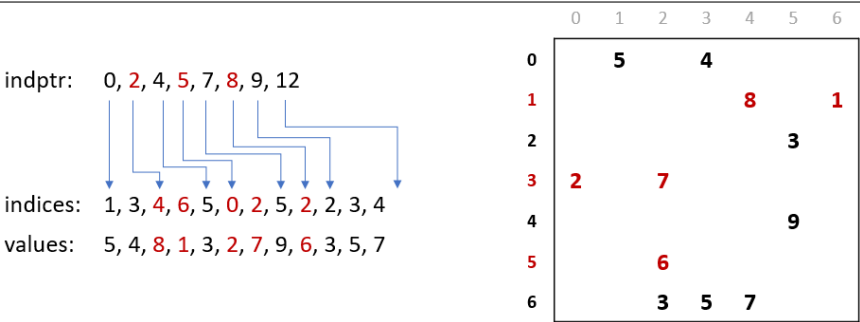


Figure B.1: Data layout for CSR format.

### 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.

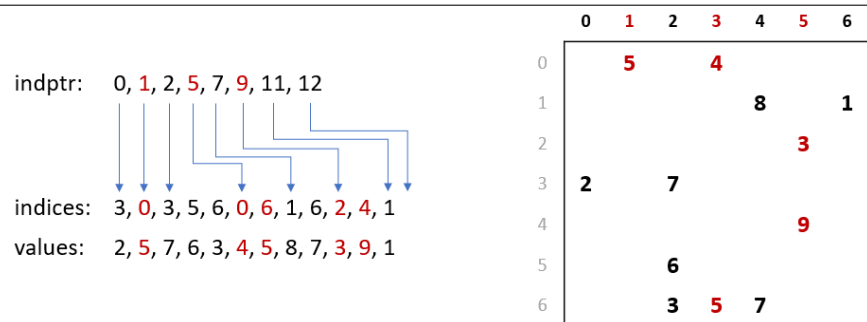


Figure B.2: Data layout for CSC format.

### 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.

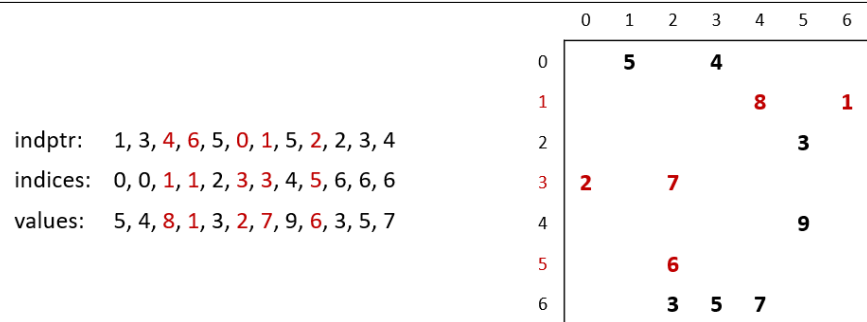


Figure B.3: Data layout for COO format.



7639 **Appendix C**

7640 **Examples**

## C.1 Example: Level breadth-first search (BFS) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS traversal
9   * of the graph and sets  $v[i]$  to the level in which vertex  $i$  is visited ( $v[s] == 1$ ).
10  * If  $i$  is not reachable from  $s$ , then  $v[i] = 0$ . (Vector  $v$  should be empty on input.)
11  */
12  GrB_Info BFS(GrB_Vector *v, GrB_Matrix A, GrB_Index s)
13  {
14      GrB_Index n;
15      GrB_Matrix_nrows(&n,A);           //  $n = \#$  of rows of  $A$ 
16
17      GrB_Vector_new(v,GrB_INT32,n);     // Vector<int32_t>  $v(n)$ 
18
19      GrB_Vector q;                      // vertices visited in each level
20      GrB_Vector_new(&q,GrB_BOOL,n);     // Vector<bool>  $q(n)$ 
21      GrB_Vector_setElement(q,(bool)true,s); //  $q[s] = \text{true}$ , false everywhere else
22
23      /*
24       * BFS traversal and label the vertices.
25       */
26      int32_t d = 0;                     //  $d = \text{level in BFS traversal}$ 
27      bool succ = false;                  //  $\text{succ} == \text{true}$  when some successor found
28      do {
29          ++d;                            // next level (start with 1)
30          GrB_assign(*v,q,GrB_NULL,d,GrB_ALL,n,GrB_NULL); //  $v[q] = d$ 
31          GrB_vxm(q,*v,GrB_NULL,GrB_LOR_LAND_SEMIRING_BOOL,
32                  q,A,GrB_DESC_RC);       //  $q[!v] = q \parallel A$ ; finds all the
33                                          // unvisited successors from current  $q$ 
34          GrB_reduce(&succ,GrB_NULL,GrB_LOR_MONOID_BOOL,
35                    q,GrB_NULL);           //  $\text{succ} = \parallel(q)$ 
36      } while (succ);                     // if there is no successor in  $q$ , we are done.
37
38      GrB_free(&q);                       //  $q$  vector no longer needed
39
40      return GrB_SUCCESS;
41  }

```

## C.2 Example: Level BFS in GraphBLAS using apply

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS traversal
9   * of the graph and sets  $v[i]$  to the level in which vertex  $i$  is visited ( $v[s] == 1$ ).
10  * If  $i$  is not reachable from  $s$ , then  $v[i]$  does not have a stored element.
11  * Vector  $v$  should be uninitialized on input.
12  */
13  GrB_Info BFS(GrB_Vector *v, const GrB_Matrix A, GrB_Index s)
14  {
15      GrB_Index n;
16      GrB_Matrix_nrows(&n,A);           //  $n = \#$  of rows of  $A$ 
17
18      GrB_Vector_new(v,GrB_INT32,n);     // Vector<int32_t>  $v(n) = 0$ 
19
20      GrB_Vector q;                     // vertices visited in each level
21      GrB_Vector_new(&q,GrB_BOOL,n);    // Vector<bool>  $q(n) = \text{false}$ 
22      GrB_Vector_setElement(q,(bool)true,s); //  $q[s] = \text{true}$ , false everywhere else
23
24      /*
25       * BFS traversal and label the vertices.
26       */
27      int32_t level = 0;                // level = depth in BFS traversal
28      GrB_Index nvals;
29      do {
30          ++level;                      // next level (start with 1)
31          GrB_apply(*v,GrB_NULL,GrB_PLUS_INT32,
32                  GrB_SECOND_INT32,q,level,GrB_NULL); //  $v[q] = \text{level}$ 
33          GrB_vxm(q,*v,GrB_NULL,GrB_LOR_LAND_SEMIRING_BOOL,
34                  q,A,GrB_DESC_RC);    //  $q[!v] = q \vee A$ ; finds all the
35                                      // unvisited successors from current  $q$ 
36          GrB_Vector_nvals(&nvals, q);
37      } while (nvals);                 // if there is no successor in  $q$ , we are done.
38
39      GrB_free(&q);                    //  $q$  vector no longer needed
40
41      return GrB_SUCCESS;
42  }

```

## C.3 Example: Parent BFS in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a binary  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS
9   * traversal of the graph and sets  $parents[i]$  to the index of vertex  $i$ 's parent.
10  * The parent of the root vertex,  $s$ , will be set to itself ( $parents[s] = s$ ). If
11  * vertex  $i$  is not reachable from  $s$ ,  $parents[i]$  will not contain a stored value.
12  */
13  GrB_Info BFS(GrB_Vector *parents, const GrB_Matrix A, GrB_Index s)
14  {
15      GrB_Index N;
16      GrB_Matrix_nrows(&N, A);           //  $N = \#$  vertices
17
18      GrB_Vector_new(parents, GrB_UINT64, N);
19      GrB_Vector_setElement(*parents, s, s);           //  $parents[s] = s$ 
20
21      GrB_Vector wavefront;
22      GrB_Vector_new(&wavefront, GrB_UINT64, N);
23      GrB_Vector_setElement(wavefront, 1UL, s);       //  $wavefront[s] = 1$ 
24
25      /*
26       * BFS traversal and label the vertices.
27       */
28      GrB_Index nvals;
29      GrB_Vector_nvals(&nvals, wavefront);
30
31      while (nvals > 0)
32      {
33          // convert all stored values in wavefront to their 0-based index
34          GrB_apply(wavefront, GrB_NULL, GrB_NULL, GrB_ROWINDEX_INT64,
35                  wavefront, 0UL, GrB_NULL);
36
37          // "FIRST" because left-multiplying wavefront rows. Masking out the parent
38          // list ensures wavefront values do not overwrite parents already stored.
39          GrB_vxm(wavefront, *parents, GrB_NULL, GrB_MIN_FIRST_SEMIRING_UINT64,
40                  wavefront, A, GrB_DESC_RSC);
41
42          // Don't need to mask here since we did it in vxm. Merges new parents in
43          // current wavefront with existing parents:  $parents += wavefront$ 
44          GrB_apply(*parents, GrB_NULL, GrB_PLUS_UINT64,
45                  GrB_IDENTITY_UINT64, wavefront, GrB_NULL);
46
47          GrB_Vector_nvals(&nvals, wavefront);
48      }
49
50      GrB_free(&wavefront);
51
52      return GrB_SUCCESS;
53  }

```

## C.4 Example: Betweenness centrality (BC) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ ,
9   * compute the BC-metric vector  $\delta$ , which should be empty on input.
10  */
11  GrB_Info BC(GrB_Vector *delta, GrB_Matrix A, GrB_Index s)
12  {
13      GrB_Index n;
14      GrB_Matrix_nrows(&n,A);                      //  $n = \#$  of vertices in graph
15
16      GrB_Vector_new(delta, GrB_FP32, n);           // Vector<float>  $\delta(n)$ 
17
18      GrB_Matrix sigma;
19      GrB_Matrix_new(&sigma, GrB_INT32, n, n);      // Matrix<int32_t>  $\sigma(n,n)$ 
20                                                    //  $\sigma[d,k] = \#$  shortest paths to node  $k$  at level  $d$ 
21
22      GrB_Vector q;
23      GrB_Vector_new(&q, GrB_INT32, n);             // Vector<int32_t>  $q(n)$  of path counts
24      GrB_Vector_setElement(q, 1, s);               //  $q[s] = 1$ 
25
26      GrB_Vector p;
27      GrB_Vector_dup(&p, q);                        // Vector<int32_t>  $p(n)$  shortest path counts so far
28                                                    //  $p = q$ 
29
30      GrB_vxm(q, p, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
31              q, A, GrB_DESC_RC);                  // get the first set of out neighbors
32
33      /*
34       * BFS phase
35       */
36      GrB_Index d = 0;                              // BFS level number
37      int32_t sum = 0;                              // sum == 0 when BFS phase is complete
38
39      do {
40          GrB_assign(sigma, GrB_NULL, GrB_NULL, q, d, GrB_ALL, n, GrB_NULL); //  $\sigma[d,:] = q$ 
41          GrB_eWiseAdd(p, GrB_NULL, GrB_NULL, GrB_PLUS_INT32, p, q, GrB_NULL); // accum path counts on this level
42          GrB_vxm(q, p, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
43                  q, A, GrB_DESC_RC);              //  $q = \#$  paths to nodes reachable
44                                                    // from current level
45          GrB_reduce(&sum, GrB_NULL, GrB_PLUS_MONOID_INT32, q, GrB_NULL); // sum path counts at this level
46          ++d;
47      } while (sum);
48
49      /*
50       * BC computation phase
51       * ( $t1, t2, t3, t4$ ) are temporary vectors
52       */
53      GrB_Vector t1; GrB_Vector_new(&t1, GrB_FP32, n);
54      GrB_Vector t2; GrB_Vector_new(&t2, GrB_FP32, n);
55      GrB_Vector t3; GrB_Vector_new(&t3, GrB_FP32, n);
56      GrB_Vector t4; GrB_Vector_new(&t4, GrB_FP32, n);
57
58      for(int i=d-1; i>0; i--)
59      {
60          GrB_assign(t1, GrB_NULL, GrB_NULL, 1.0f, GrB_ALL, n, GrB_NULL); //  $t1 = 1 + \delta$ 
61          GrB_eWiseAdd(t1, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, t1, *delta, GrB_NULL);
62          GrB_extract(t2, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i, GrB_DESC_T0); //  $t2 = \sigma[i,:]$ 
63          GrB_eWiseMult(t2, GrB_NULL, GrB_NULL, GrB_DIV_FP32, t1, t2, GrB_NULL); //  $t2 = (1 + \delta) / \sigma[i,:]$ 
64          GrB_mvx(t3, GrB_NULL, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32,
65                  // add contributions made by

```

```

63         A, t2, GrB_NULL);
64     GrB_extract(t4, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i-1, GrB_DESC_T0); // t4 = sigma[i-1,:]
65     GrB_eWiseMult(t4, GrB_NULL, GrB_NULL, GrB_TIMES_FP32, t4, t3, GrB_NULL); // t4 = sigma[i-1,:]*t3
66     GrB_eWiseAdd(*delta, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, *delta, t4, GrB_NULL); // accumulate into delta
67 }
68
69 GrB_free(&sigma);
70 GrB_free(&q); GrB_free(&p);
71 GrB_free(&t1); GrB_free(&t2); GrB_free(&t3); GrB_free(&t4);
72
73 return GrB_SUCCESS;
74 }

```

## C.5 Example: Batched BC in GraphBLAS

```

1  #include <stdlib.h>
2  #include "GraphBLAS.h" // in addition to other required C headers
3
4  // Compute partial BC metric for a subset of source vertices, s, in graph A
5  GrB_Info BC_update(GrB_Vector *delta, GrB_Matrix A, GrB_Index *s, GrB_Index nsver)
6  {
7      GrB_Index n;
8      GrB_Matrix_nrows(&n, A); // n = # of vertices in graph
9      GrB_Vector_new(delta, GrB_FP32, n); // Vector<float> delta(n)
10
11     // index and value arrays needed to build numsp
12     GrB_Index *i_nsver = (GrB_Index*) malloc(sizeof(GrB_Index)*nsver);
13     int32_t *ones = (int32_t*) malloc(sizeof(int32_t)*nsver);
14     for(int i=0; i<nsver; ++i) {
15         i_nsver[i] = i;
16         ones[i] = 1;
17     }
18
19     // numsp: structure holds the number of shortest paths for each node and starting vertex
20     // discovered so far. Initialized to source vertices: numsp[s[i],i]=1, i=[0,nsver)
21     GrB_Matrix numsp;
22     GrB_Matrix_new(&numsp, GrB_INT32, n, nsver);
23     GrB_Matrix_build(numsp, s, i_nsver, ones, nsver, GrB_PLUS_INT32);
24     free(i_nsver); free(ones);
25
26     // frontier: Holds the current frontier where values are path counts.
27     // Initialized to out vertices of each source node in s.
28     GrB_Matrix frontier;
29     GrB_Matrix_new(&frontier, GrB_INT32, n, nsver);
30     GrB_extract(frontier, numsp, GrB_NULL, A, GrB_ALL, n, s, nsver, GrB_DESC_RCT0);
31
32     // sigma: stores frontier information for each level of BFS phase. The memory
33     // for an entry in sigmas is only allocated within the do-while loop if needed.
34     // n is an upper bound on diameter.
35     GrB_Matrix *sigmas = (GrB_Matrix*) malloc(sizeof(GrB_Matrix)*n);
36
37     int32_t d = 0; // BFS level number
38     GrB_Index nvals = 0; // nvals == 0 when BFS phase is complete
39
40     // ----- The BFS phase (forward sweep) -----
41     do {
42         // sigmas[d](:,s) = d^th level frontier from source vertex s
43         GrB_Matrix_new(&(sigmas[d]), GrB_BOOL, n, nsver);
44
45         GrB_apply(sigmas[d], GrB_NULL, GrB_NULL,
46                 GrB_IDENTITY_BOOL, frontier, GrB_NULL); // sigmas[d](:,:) = (Boolean) frontier
47         GrB_eWiseAdd(numsp, GrB_NULL, GrB_NULL, GrB_PLUS_INT32,
48                     numsp, frontier, GrB_NULL); // numsp += frontier (accum path counts)
49         GrB_mxm(frontier, numsp, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
50                 A, frontier, GrB_DESC_RCT0); // f<!numsp> = A' +.* f (update frontier)
51         GrB_Matrix_nvals(&nvals, frontier); // number of nodes in frontier at this level
52         d++;
53     } while (nvals);
54
55     // nspinv: the inverse of the number of shortest paths for each node and starting vertex.
56     GrB_Matrix nspinv;
57     GrB_Matrix_new(&nspinv, GrB_FP32, n, nsver);
58     GrB_apply(nspinv, GrB_NULL, GrB_NULL,
59              GrB_MINV_FP32, numsp, GrB_NULL); // nspinv = 1./numsp
60
61     // bcu: BC updates for each vertex for each starting vertex in s
62     GrB_Matrix bcu;

```

```

63 GrB_Matrix_new(&bcu, GrB_FP32, n, nsver);
64 GrB_assign(bcu, GrB_NULL, GrB_NULL,
65           1.0f, GrB_ALL, n, GrB_ALL, nsver, GrB_NULL); // filled with 1 to avoid sparsity issues
66
67 GrB_Matrix w; // temporary workspace matrix
68 GrB_Matrix_new(&w, GrB_FP32, n, nsver);
69
70 // ----- Tally phase (backward sweep) -----
71 for (int i=d-1; i>0; i--) {
72     GrB_eWiseMult(w, sigmas[i], GrB_NULL,
73                 GrB_TIMES_FP32, bcu, nspinv, GrB_DESC_R); // w<sigmas[i]>=(1 ./ nsp).*bcu
74
75     // add contributions by successors and mask with that BFS level's frontier
76     GrB_mxm(w, sigmas[i-1], GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32,
77            A, w, GrB_DESC_R); // w<sigmas[i-1]> = (A +.* w)
78     GrB_eWiseMult(bcu, GrB_NULL, GrB_PLUS_FP32, GrB_TIMES_FP32,
79                 w, numsp, GrB_NULL); // bcu += w .* numsp
80 }
81
82 // row reduce bcu and subtract "nsver" from every entry to account
83 // for 1 extra value per bcu row element.
84 GrB_reduce(*delta, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, bcu, GrB_NULL);
85 GrB_apply(*delta, GrB_NULL, GrB_NULL, GrB_MINUS_FP32, *delta, (float)nsver, GrB_NULL);
86
87 // Release resources
88 for (int i=0; i<d; i++) {
89     GrB_free(&(sigmas[i]));
90 }
91 free(sigmas);
92
93 GrB_free(&frontier); GrB_free(&numsp);
94 GrB_free(&nspinv); GrB_free(&bcu); GrB_free(&w);
95
96 return GrB_SUCCESS;
97 }

```



## C.6 Example: Maximal independent set (MIS) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  // Assign a random number to each element scaled by the inverse of the node's degree.
8  // This will increase the probability that low degree nodes are selected and larger
9  // sets are selected.
10 void setRandom(void *out, const void *in)
11 {
12     uint32_t degree = *(uint32_t*)in;
13     *(float*)out = (0.0001f + random()/(1. + 2.*degree)); // add 1 to prevent divide by zero
14 }
15
16 /*
17  * A variant of Luby's randomized algorithm [Luby 1985].
18  *
19  * Given a numeric n x n adjacency matrix A of an unweighted and undirected graph (where
20  * the value true represents an edge), compute a maximal set of independent vertices and
21  * return it in a boolean n-vector, 'iset' where set[i] == true implies vertex i is a member
22  * of the set (the iset vector should be uninitialized on input.)
23  */
24 GrB_Info MIS(GrB_Vector *iset, const GrB_Matrix A)
25 {
26     GrB_Index n;
27     GrB_Matrix_nrows(&n,A); // n = # of rows of A
28
29     GrB_Vector prob; // holds random probabilities for each node
30     GrB_Vector neighbor_max; // holds value of max neighbor probability
31     GrB_Vector new_members; // holds set of new members to iset
32     GrB_Vector new_neighbors; // holds set of new neighbors to new iset mbrs.
33     GrB_Vector candidates; // candidate members to iset
34
35     GrB_Vector_new(&prob,GrB_FP32,n);
36     GrB_Vector_new(&neighbor_max,GrB_FP32,n);
37     GrB_Vector_new(&new_members,GrB_BOOL,n);
38     GrB_Vector_new(&new_neighbors,GrB_BOOL,n);
39     GrB_Vector_new(&candidates,GrB_BOOL,n);
40     GrB_Vector_new(iset,GrB_BOOL,n); // Initialize independent set vector, bool
41
42     GrB_UnaryOp set_random;
43     GrB_UnaryOp_new(&set_random,setRandom,GrB_FP32,GrB_UINT32);
44
45     // compute the degree of each vertex.
46     GrB_Vector degrees;
47     GrB_Vector_new(&degrees,GrB_FP64,n);
48     GrB_reduce(degrees,GrB_NULL,GrB_NULL,GrB_PLUS_FP64,A,GrB_NULL);
49
50     // Isolated vertices are not candidates: candidates[degrees != 0] = true
51     GrB_assign(candidates,degrees,GrB_NULL,true,GrB_ALL,n,GrB_NULL);
52
53     // add all singletons to iset: iset[degree == 0] = 1
54     GrB_assign(*iset,degrees,GrB_NULL,true,GrB_ALL,n,GrB_DESC_RC);
55
56     // Iterate while there are candidates to check.
57     GrB_Index nvals;
58     GrB_Vector_nvals(&nvals,candidates);
59     while (nvals > 0) {
60         // compute a random probability scaled by inverse of degree
61         GrB_apply(prob,candidates,GrB_NULL,set_random,degrees,GrB_DESC_R);
62     }

```

```

63 // compute the max probability of all neighbors
64 GrB_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 }
```