

The GraphBLAS C API Specification [†]:

Version 2.1

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

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

Generated on 2023/11/01 at 10:01:35 EDT

[†]Based on *GraphBLAS Mathematics* by Jeremy Kepner

7 Copyright © 2017-2023 Carnegie Mellon University, The Regents of the University of California,
8 through Lawrence Berkeley National Laboratory (subject to receipt of any required approvals from
9 the U.S. Dept. of Energy), the Regents of the University of California (U.C. Davis and U.C.
10 Berkeley), Intel Corporation, International Business Machines Corporation, and Massachusetts
11 Institute of Technology Lincoln Laboratory.

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

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

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

Contents

27	List of Tables	9
28	List of Figures	11
29	Acknowledgments	12
30	1 Introduction	15
31	2 Basic concepts	17
32	2.1 Glossary	17
33	2.1.1 GraphBLAS API basic definitions	17
34	2.1.2 GraphBLAS objects and their structure	18
35	2.1.3 Algebraic structures used in the GraphBLAS	19
36	2.1.4 The execution of an application using the GraphBLAS C API	20
37	2.1.5 GraphBLAS methods: behaviors and error conditions	21
38	2.2 Notation	23
39	2.3 Mathematical foundations	24
40	2.4 GraphBLAS opaque objects	25
41	2.5 Execution model	26
42	2.5.1 Execution modes	27
43	2.5.2 Multi-threaded execution	28
44	2.6 Error model	30
45	3 Objects	33
46	3.1 Enumerations for <code>init()</code> and <code>wait()</code>	33
47	3.2 Indices, index arrays, and scalar arrays	33
48	3.3 Types (domains)	34

49	3.4	Algebraic objects, operators and associated functions	35
50	3.4.1	Operators	36
51	3.4.2	Monoids	41
52	3.4.3	Semirings	41
53	3.5	Collections	45
54	3.5.1	Scalars	45
55	3.5.2	Vectors	45
56	3.5.3	Matrices	46
57	3.5.3.1	External matrix formats	46
58	3.5.4	Masks	46
59	3.6	Descriptors	47
60	3.7	Fields	48
61	3.7.1	Input Types	51
62	3.7.1.1	INT32 Handling	51
63	3.7.1.2	GrB_Scalar Handling	51
64	3.7.1.3	String (char*) Handling	51
65	3.7.1.4	void* Handling	51
66	3.7.1.5	SIZE Handling	51
67	3.7.2	Hints	52
68	3.7.3	GrB_NAME	52
69	3.8	GrB_Info return values	54
70	4	Methods	57
71	4.1	Context methods	57
72	4.1.1	init: Initialize a GraphBLAS context	57
73	4.1.2	finalize: Finalize a GraphBLAS context	58
74	4.1.3	getVersion: Get the version number of the standard.	59
75	4.2	Object methods	59
76	4.2.1	Get and Set methods	60
77	4.2.1.1	get: Query the value of an object	60
78	4.2.1.2	set: Set field of an object	60

79	4.2.2	Algebra methods	61
80	4.2.2.1	Type_new: Construct a new GraphBLAS (user-defined) type	61
81	4.2.2.2	UnaryOp_new: Construct a new GraphBLAS unary operator	62
82	4.2.2.3	BinaryOp_new: Construct a new GraphBLAS binary operator	64
83	4.2.2.4	Monoid_new: Construct a new GraphBLAS monoid	66
84	4.2.2.5	Semiring_new: Construct a new GraphBLAS semiring	67
85	4.2.2.6	IndexUnaryOp_new: Construct a new GraphBLAS index unary op-	
86		erator [Scott: NEW CONTENT]	68
87	4.2.3	Scalar methods	70
88	4.2.3.1	Scalar_new: Construct a new scalar	70
89	4.2.3.2	Scalar_dup: Construct a copy of a GraphBLAS scalar	71
90	4.2.3.3	Scalar_clear: Clear/remove a stored value from a scalar	72
91	4.2.3.4	Scalar_nvals: Number of stored elements in a scalar	73
92	4.2.3.5	Scalar_setElement: Set the single element in a scalar	74
93	4.2.3.6	Scalar_extractElement: Extract a single element from a scalar. . . .	75
94	4.2.4	Vector methods	76
95	4.2.4.1	Vector_new: Construct new vector	76
96	4.2.4.2	Vector_dup: Construct a copy of a GraphBLAS vector	77
97	4.2.4.3	Vector_resize: Resize a vector	78
98	4.2.4.4	Vector_clear: Clear a vector	79
99	4.2.4.5	Vector_size: Size of a vector	80
100	4.2.4.6	Vector_nvals: Number of stored elements in a vector	81
101	4.2.4.7	Vector_build: Store elements from tuples into a vector	82
102	4.2.4.8	Vector_setElement: Set a single element in a vector	84
103	4.2.4.9	Vector_removeElement: Remove an element from a vector	86
104	4.2.4.10	Vector_extractElement: Extract a single element from a vector. . . .	87
105	4.2.4.11	Vector_extractTuples: Extract tuples from a vector	89
106	4.2.5	Matrix methods	90
107	4.2.5.1	Matrix_new: Construct new matrix	90
108	4.2.5.2	Matrix_dup: Construct a copy of a GraphBLAS matrix	92
109	4.2.5.3	Matrix_diag: Construct a diagonal GraphBLAS matrix	93

110	4.2.5.4	Matrix_resize: Resize a matrix	94
111	4.2.5.5	Matrix_clear: Clear a matrix	95
112	4.2.5.6	Matrix_nrows: Number of rows in a matrix	96
113	4.2.5.7	Matrix_ncols: Number of columns in a matrix	96
114	4.2.5.8	Matrix_nvals: Number of stored elements in a matrix	97
115	4.2.5.9	Matrix_build: Store elements from tuples into a matrix	98
116	4.2.5.10	Matrix_setElement: Set a single element in matrix	100
117	4.2.5.11	Matrix_removeElement: Remove an element from a matrix	102
118	4.2.5.12	Matrix_extractElement: Extract a single element from a matrix . . .	103
119	4.2.5.13	Matrix_extractTuples: Extract tuples from a matrix	105
120	4.2.5.14	Matrix_exportHint: Provide a hint as to which storage format might be most efficient for exporting a matrix	107
122	4.2.5.15	Matrix_exportSize: Return the array sizes necessary to export a GraphBLAS matrix object	108
124	4.2.5.16	Matrix_export: Export a GraphBLAS matrix to a pre-defined format	109
125	4.2.5.17	Matrix_import: Import a matrix into a GraphBLAS object	111
126	4.2.5.18	Matrix_serializeSize: Compute the serialize buffer size	113
127	4.2.5.19	Matrix_serialize: Serialize a GraphBLAS matrix.	114
128	4.2.5.20	Matrix_deserialize: Deserialize a GraphBLAS matrix.	115
129	4.2.6	Descriptor methods	116
130	4.2.6.1	Descriptor_new: Create new descriptor	116
131	4.2.6.2	Descriptor_set: Set content of descriptor	117
132	4.2.7	free: Destroy an object and release its resources	118
133	4.2.8	wait: Return once an object is either <i>complete</i> or <i>materialized</i>	120
134	4.2.9	error: Retrieve an error string	121
135	4.3	GraphBLAS operations	122
136	4.3.1	mxm: Matrix-matrix multiply	126
137	4.3.2	vxm: Vector-matrix multiply	131
138	4.3.3	mxv: Matrix-vector multiply	135
139	4.3.4	eWiseMult: Element-wise multiplication	139
140	4.3.4.1	eWiseMult: Vector variant	140

141	4.3.4.2	eWiseMult: Matrix variant	144
142	4.3.5	eWiseAdd: Element-wise addition	149
143	4.3.5.1	eWiseAdd: Vector variant	150
144	4.3.5.2	eWiseAdd: Matrix variant	154
145	4.3.6	extract: Selecting sub-graphs	160
146	4.3.6.1	extract: Standard vector variant	160
147	4.3.6.2	extract: Standard matrix variant	164
148	4.3.6.3	extract: Column (and row) variant	169
149	4.3.7	assign: Modifying sub-graphs	174
150	4.3.7.1	assign: Standard vector variant	174
151	4.3.7.2	assign: Standard matrix variant	179
152	4.3.7.3	assign: Column variant	185
153	4.3.7.4	assign: Row variant	190
154	4.3.7.5	assign: Constant vector variant[Scott: NEW CONTENT]	196
155	4.3.7.6	assign: Constant matrix variant[Scott: NEW CONTENT]	201
156	4.3.8	apply: Apply a function to the elements of an object	207
157	4.3.8.1	apply: Vector variant	207
158	4.3.8.2	apply: Matrix variant	212
159	4.3.8.3	apply: Vector-BinaryOp variants[Scott: NEW CONTENT]	216
160	4.3.8.4	apply: Matrix-BinaryOp variants[Scott: NEW CONTENT]	222
161	4.3.8.5	apply: Vector index unary operator variant[Scott: NEW CONTENT]	228
162	4.3.8.6	apply: Matrix index unary operator variant[Scott: NEW CONTENT]	233
163	4.3.9	select:	238
164	4.3.9.1	select: Vector variant[Scott: NEW CONTENT]	238
165	4.3.9.2	select: Matrix variant[Scott: NEW CONTENT]	243
166	4.3.10	reduce: Perform a reduction across the elements of an object	249
167	4.3.10.1	reduce: Standard matrix to vector variant	249
168	4.3.10.2	reduce: Vector-scalar variant[Scott: NEW CONTENT]	253
169	4.3.10.3	reduce: Matrix-scalar variant[Scott: NEW CONTENT]	257
170	4.3.11	transpose: Transpose rows and columns of a matrix	260

171	4.3.12 kronecker: Kronecker product of two matrices	264
172	5 Nonpolymorphic interface [Scott: NEW CONTENT]	271
173	A Revision history	285
174	B Non-opaque data format definitions	291
175	B.1 GrB_Format: Specify the format for input/output of a GraphBLAS matrix.	291
176	B.1.1 GrB_CSR_FORMAT	291
177	B.1.2 GrB_CSC_FORMAT	292
178	B.1.3 GrB_COO_FORMAT	292
179	C Examples	293
180	C.1 Example: Level breadth-first search (BFS) in GraphBLAS	294
181	C.2 Example: Level BFS in GraphBLAS using apply	295
182	C.3 Example: Parent BFS in GraphBLAS	296
183	C.4 Example: Betweenness centrality (BC) in GraphBLAS	297
184	C.5 Example: Batched BC in GraphBLAS	299
185	C.6 Example: Maximal independent set (MIS) in GraphBLAS	301
186	C.7 Example: Counting triangles in GraphBLAS	303

List of Tables

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

213	4.1	A mathematical notation for the fundamental GraphBLAS operations supported in	
214		this specification.	123
215	5.1	Long-name, nonpolymorphic form of GraphBLAS methods.	271
216	5.2	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	272
217	5.3	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	273
218	5.4	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	274
219	5.5	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	275
220	5.6	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	276
221	5.7	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	277
222	5.8	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	278
223	5.9	Long-name, nonpolymorphic form of GraphBLAS methods (continued).[Scott: NEW	
224		CONTENT]	279
225	5.10	Long-name, nonpolymorphic form of GraphBLAS methods (continued).[Scott: NEW	
226		CONTENT]	280
227	5.11	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	281
228	5.12	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	282
229	5.13	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	283

230 List of Figures

231	3.1 Hierarchy of algebraic object classes in GraphBLAS.	45
232	4.1 Flowchart for the GraphBLAS operations.	124
233	B.1 Data layout for CSR format.	291
234	B.2 Data layout for CSC format.	292
235	B.3 Data layout for COO format.	292

Acknowledgments

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

Those who served as C API Subcommittee members for GraphBLAS 2.1 are (in alphabetical order):

- Raye Kimmerer (MIT)
- Jim Kitchen (Anaconda)
- Manoj Kumar (?)
- Timothy G. Mattson (Intel Corporation)
- Erik Welch (Nvidia Corporation)

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

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

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

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

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

- NSF Graduate Research Fellowship under Grant No. DGE 1752814 and by the NSF under Award No. 1823034 with the University of California, Berkeley
- The Department of Energy Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231

- 263 • Intel Corporation
 - 264 • Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon Uni-
265 versity for the operation of the Software Engineering Institute [DM-0003727, DM19-0929,
266 DM21-0090]
 - 267 • International Business Machines Corporation
- 268 The following people provided valuable input and feedback during the development of the spec-
269 ification (in alphabetical order): David Bader, Hollen Barmer, Bob Cook, Tim Davis, Jeremy
270 Kepner, Jim Kitchen, Peter Kogge, Manoj Kumar, Roi Lipman, Andrew Mellinger, Maxim Nau-
271 mov, Nancy M. Ott, Michel Pelletier, Gabor Szarnyas, Ping Tak Peter Tang, Erik Welch, Michael
272 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*

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

- 492 • *dimension compatible*: GraphBLAS objects (matrices and vectors) that are passed as param-
493 eters to a GraphBLAS method are dimension (or shape) compatible if they have the correct
494 number of dimensions and sizes for each dimension to satisfy the rules of the mathematical def-
495 inition of the operation associated with the method. If any *dimension compatibility* rule above
496 is violated, execution of the GraphBLAS method ends and the GrB_DIMENSION_MISMATCH
497 error is returned.
- 498 • *domain compatible*: Two domains for which values from one domain can be cast to values in
499 the other domain as per the rules of the C language. In particular, domains from Table 3.2
500 are all compatible with each other, and a domain from a user-defined type is only compatible
501 with itself. If any *domain compatibility* rule above is violated, execution of the GraphBLAS
502 method ends and the GrB_DOMAIN_MISMATCH error is returned.

2.2 Notation

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

2.3 Mathematical foundations

Graphs can be represented in terms of matrices. The values stored in these matrices correspond to attributes (often weights) of edges in the graph.¹ 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

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

Table 2.1: Types of GraphBLAS opaque objects.

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

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

2.4 GraphBLAS opaque objects

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

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

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

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

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

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

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

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

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

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

2.5 Execution model

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

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

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

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

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

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

2.5.1 Execution modes

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

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

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

GrB_init() function. This function takes a single argument of type GrB_Mode with values shown in Table 3.1(a).

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

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

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

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

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

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

2.5.2 Multi-threaded execution

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

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

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

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

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

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

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

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

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

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

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

2.6 Error model

All GraphBLAS methods return a value of type `GrB_Info` (an enum) to provide information available to the system at the time the method returns. The returned value will be one of the defined values shown in Table 3.16. The return values fall into three groups: informational, API errors, and execution errors. While API and execution errors take on negative values, informational return values listed in Table 3.16(a) are non-negative and include `GrB_SUCCESS` (a value of 0) and `GrB_NO_VALUE`.

An API error (listed in Table 3.16(b)) means that a GraphBLAS method was called with parameters that violate the rules for that method. These errors are restricted to those that can be determined by inspecting the dimensions and domains of GraphBLAS objects, GraphBLAS operators, or the values of scalar parameters fixed at the time a method is called. API errors are deterministic and consistent across platforms and implementations. API errors are never deferred, even in nonblocking mode. That is, if a method is called in a manner that would generate an API error, it always returns with the appropriate API error value. If a GraphBLAS method returns with an API error, it is guaranteed that none of the arguments to the method (or any other program data) have been modified. The informational return value, `GrB_NO_VALUE`, is also deterministic and never deferred in nonblocking mode.

Execution errors (listed in Table 3.16(c)) indicate that something went wrong during the execution of a legal GraphBLAS method invocation. Their occurrence may depend on specifics of the execution environment and data values being manipulated. This does not mean that execution errors are the fault of the GraphBLAS implementation. For example, a memory leak could arise from an error in an application’s source code (a “program error”), but it may manifest itself in different points of a program’s execution (or not at all) depending on the platform, problem size, or what else is running at that time. Index out-of-bounds errors, for example, always indicate a program error.

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

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

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

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

763 After a call to any GraphBLAS method that modifies an opaque object, the program can re-
764 trieve additional error information (beyond the error code returned by the method) though a call
765 to the function `GrB_error()`, passing the method's output object as described in Section 4.2.9.
766 The function returns a pointer to a NULL-terminated string, and the contents of that string are
767 implementation-dependent. In particular, a null string (not a NULL pointer) is always a valid error
768 string. `GrB_error()` is a thread-safe function, in the sense that multiple threads can call it simul-
769 taneously and each will get its own error string back, referring to the object passed as an input
770 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:

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

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

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

807 3.3 Types (domains)

808 In GraphBLAS, domains correspond to the valid values for types from the host language (in our
809 case, the C programming language). GraphBLAS defines a number of operators that take elements
810 from one or more domains and produce elements of a (possibly) different domain. GraphBLAS
811 also defines three kinds of collections: matrices, vectors and scalars. For any given collection, the
812 elements of the collection belong to a *domain*, which is the set of valid values for the elements. For
813 any variable or object V in GraphBLAS we denote as $\mathbf{D}(V)$ the domain of V , that is, the set of
814 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).

<code>GrB_Type</code>	<code>GrB_Type_Code</code>	Suffix	C type	Domain
-	<code>GrB_UDT_CODE=0</code>	UDT	-	-
<code>GrB_BOOL</code>	<code>GrB_BOOL_CODE=1</code>	BOOL	<code>bool</code>	$\{\text{false}, \text{true}\}$
<code>GrB_INT8</code>	<code>GrB_INT8_CODE=2</code>	INT8	<code>int8_t</code>	$\mathbb{Z} \cap [-2^7, 2^7)$
<code>GrB_UINT8</code>	<code>GrB_UINT8_CODE=3</code>	UINT8	<code>uint8_t</code>	$\mathbb{Z} \cap [0, 2^8)$
<code>GrB_INT16</code>	<code>GrB_INT16_CODE=4</code>	INT16	<code>int16_t</code>	$\mathbb{Z} \cap [-2^{15}, 2^{15})$
<code>GrB_UINT16</code>	<code>GrB_UINT16_CODE=5</code>	UINT16	<code>uint16_t</code>	$\mathbb{Z} \cap [0, 2^{16})$
<code>GrB_INT32</code>	<code>GrB_INT32_CODE=6</code>	INT32	<code>int32_t</code>	$\mathbb{Z} \cap [-2^{31}, 2^{31})$
<code>GrB_UINT32</code>	<code>GrB_UINT32_CODE=7</code>	UINT32	<code>uint32_t</code>	$\mathbb{Z} \cap [0, 2^{32})$
<code>GrB_INT64</code>	<code>GrB_INT64_CODE=8</code>	INT64	<code>int64_t</code>	$\mathbb{Z} \cap [-2^{63}, 2^{63})$
<code>GrB_UINT64</code>	<code>GrB_UINT64_CODE=9</code>	UINT64	<code>uint64_t</code>	$\mathbb{Z} \cap [0, 2^{64})$
<code>GrB_FP32</code>	<code>GrB_FP32_CODE=10</code>	FP32	<code>float</code>	IEEE 754 binary32
<code>GrB_FP64</code>	<code>GrB_FP64_CODE=11</code>	FP64	<code>double</code>	IEEE 754 binary64

The domains for elements that can be stored in collections and operated on through GraphBLAS methods are defined by GraphBLAS objects called `GrB_Type`. The predefined types and corresponding domains used in the GraphBLAS C API are shown in Table 3.2. The Boolean type (`bool`) is defined in `stdbool.h`, the integral types (`int8_t`, `uint8_t`, `int16_t`, `uint16_t`, `int32_t`, `uint32_t`, `int64_t`, `uint64_t`) are defined in `stdint.h`, and the floating-point types (`float`, `double`) are native to the language and platform and in most cases defined by the IEEE-754 standard. UDT stands for user-defined type and is the type code returned for all objects which use a non-predefined type. Implementations which add new types should start their `GrB_Type_Codes` at 100 to avoid possible conflicts with built-in types which may be added in the future.

3.4 Algebraic objects, operators and associated functions

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

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

Some GraphBLAS operations require a monoid or semiring. A monoid contains an associative

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

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

binary operator where the input and output domains are the same. The monoid also includes an identity value of the operator. The semiring consists of a binary operator – referred to as the “times” operator – with up to three different domains (two inputs and one output) and a monoid – referred to as the “plus” operator – that is also commutative. Furthermore, the domain of the monoid must be the same as the output domain of the “times” operator.

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

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

3.4.1 Operators

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

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

(a) Properties of algebraic objects.

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

(b) Recipes for algebraic objects.

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

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

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

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

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

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

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

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

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

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

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

3.4.2 Monoids

A GraphBLAS *monoid* $M = \langle D, \odot, 0 \rangle$ is defined by a single domain D , an *associative*¹ 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*¹ 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.

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

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

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

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

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

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

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

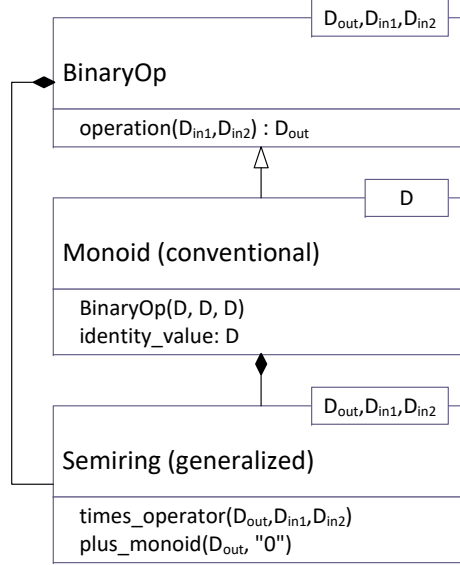


Figure 3.1: Hierarchy of algebraic object classes in GraphBLAS. GraphBLAS semirings consist of a conventional monoid with one domain for the addition function, and a binary operator with three domains for the multiplication function.

3.5 Collections

3.5.1 Scalars

A *GraphBLAS scalar*, $s = \langle D, \{\sigma\} \rangle$, is defined by a domain D , and a set of zero or one *scalar value*, σ , 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, σ , if the GraphBLAS scalar is not empty, and is undefined otherwise.

3.5.2 Vectors

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

3.5.3 Matrices

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

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

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

3.5.3.1 External matrix formats

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

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

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

3.5.4 Masks

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

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

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

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

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

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

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

3.6 Descriptors

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

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

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

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

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

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

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

3.7 Fields

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

The required *value, field* pairs available for each object are defined in 3.13. Implementations may add their own custom `GrB_Field` enum values to extend the behavior of objects and methods. A field must always be readable, but in many cases may not be writable. Such read-only fields might contain static, compile-time information such as `GrB_API_VER`, while others are determined by other operations, such as `GrB_BLOCKING_MODE` which is determined by `GrB_Init`.

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

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

Table 3.11: Descriptors are GraphBLAS objects passed as arguments to GraphBLAS operations to modify other GraphBLAS objects in the operation’s argument list. A descriptor, `desc`, has one or more (*field*, *value*) pairs indicated as `desc[GrB_Desc_Field].GrB_Desc_Value`. In this table, we define all types and literals used with descriptors.

(a) Types used with GraphBLAS descriptors.

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

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

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

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

Value Name	Value	Description
<code>GrB_DEFAULT</code>	0	The default (unset) value for each field.
<code>GrB_REPLACE</code>	1	Clear the output object before assigning computed values.
<code>GrB_COMP</code>	2	Use the complement of the associated object. When combined with <code>GrB_STRUCTURE</code> , the complement of the structure of the associated object is used without evaluating the values stored.
<code>GrB_TRAN</code>	3	Use the transpose of the associated object.
<code>GrB_STRUCTURE</code>	4	The write mask is constructed from the structure (pattern of stored values) of the associated object. The stored values are not examined.
<code>GrB_COMP_STRUCTURE</code>	6	Shorthand for both <code>GrB_COMP</code> and <code>GrB_STRUCTURE</code> .

Table 3.12: Predefined GraphBLAS descriptors. The list includes all possible descriptors, according to the current standard. Columns list the possible fields and entries list the value(s) associated with those fields for a given descriptor.

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

1020 3.7.1 Input Types

1021 Allowable types used in `GrB_get` and `GrB_set` are `INT32`, `GrB_Scalar`, `char*`, `void*`, and `SIZE`. Each
1022 `GrB_Field` is associated with exactly one of these types as defined in Table 3.13. Implementations
1023 that add additional `GrB_Fields` must document the type associated with each `GrB_Field`.

1024 3.7.1.1 INT32 Handling

1025 `INT32` types use a 32-bit signed integer type. This can be used both for numeric values as well as
1026 enumerated C types. Enumerated types must specify the numeric value for each enum, and the
1027 value specified must fit within the allowable 32-bit signed integer range.

1028 3.7.1.2 GrB_Scalar Handling

1029 When calling `GrB_get`, the user must provide an already initialized `GrB_Scalar` object to which
1030 the implementation will write a value of the correct element type. When calling `GrB_set`, the
1031 `GrB_Scalar` must not be empty, otherwise a `GrB_EMPTY_OBJECT` error is raised.

1032 3.7.1.3 String (char*) Handling

1033 When the input to `GrB_set` is a `char*` the input array is null terminated. The `GraphBLAS` imple-
1034 mentation must copy this array into internal data structures. Using `GrB_get` for strings requires
1035 two calls. First, call `GrB_get` with the field and object, but pass `size_t*` as the value argument.
1036 The implementation will return the size of the string buffer that the user must create. Second, call
1037 `GrB_get` with the field and object, this time passing a pointer to the newly created string buffer.
1038 The `GraphBLAS` implementation will write to this buffer, including a trailing null terminator. The
1039 size returned in the first call will include enough bytes for the null terminator.

1040 3.7.1.4 void* Handling

1041 When the input to `GrB_set` is a `void*`, an extra `size_t` argument is passed to indicate the size of the
1042 buffer. The `GraphBLAS` implementation must copy this many bytes from the buffer into internal
1043 data structures. Similar to reading strings, `GrB_get` must be called twice for `void*`. The first call
1044 passes `size_t*` to find the required buffer size. The user must create a buffer and then pass the
1045 pointer to `GrB_get`. The implementation will write to this buffer. No standard specification or
1046 protocol is required for the contents of `void*`. It is meant to be a mechanism to allow full freedom
1047 for `GraphBLAS` implementations with needs that cannot be handled using `INT32`, `GrB_Scalar`, or
1048 `Strings`.

1049 3.7.1.5 SIZE Handling

1050 `SIZE` types use a `size_t` type. Normally, `SIZE` is used in conjunction with `char*` and `void*` to indicate
1051 the buffer size. However, it can also be used when the actual return type is `size_t`, as is the case

1052 for the size of a Type.

1053 3.7.2 Hints

1054 Several fields are *hints* (marked H in Table 3.13). Hints are used to represent intended use cases
1055 or best guesses, but do not determine strict behavior. When `GrB_set` is called with a hint, the
1056 GraphBLAS implementation should return `GrB_SUCCESS`, but is free to use or ignore the hint.
1057 When `GrB_get` is called, the implementation should return a best guess on the correct answer. If
1058 there is no clear answer, the implementation should return `GrB_UNKNOWN`.

1059 3.7.3 GrB_NAME

1060 The `GrB_NAME` field is a special case regarding writability. All user-defined objects have a
1061 `GrB_NAME` field which defaults to an empty string. Collections and `GrB_Descriptors` may have
1062 their `GrB_NAME` set at any time. User-defined algebraic objects and `GrB_Types` may only have
1063 their `GrB_NAME` set once to a globally unique value. Attempting to set this field after it has
1064 already been set will return a `GrB_ALREADY_SET` error code.

1065 Built-in algebraic objects and `GrB_Types` have names which can be read, but not written to. The
1066 name returned will be the string form of the `GrB_Type` listed in Table 3.2 or the GraphBLAS
1067 identifier listed in Tables 3.5, 3.6, 3.7, 3.8, and 3.9. For example, the name of `GrB_BOOL` type
1068 is "`GrB_BOOL`" (8 characters) and the name of `GrB_MIN_FP64` binary op is "`GrB_MIN_FP64`" (12
1069 characters).

1070 The `GrB_NAME` of the global context is read-only and returns the name of the library implemen-
1071 tation.

Table 3.13: Field values of type GrB_Field enumeration, corresponding types, and the objects which must implement that GrB_Field. Collection refers to GrB_Matrix, GrB_Vector, and GrB_Scalar, Algebraic refers to Operators, Monoids, and Semirings, Type refers to GrB_Type, and Global refers to the GrB_Global context. All fields may be read, some may be written (denoted by W), and some are hints (denoted by H) which may be ignored by the implementation. For * see 3.7

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

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

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

3.8 GrB_Info return values

All GraphBLAS methods return a GrB_Info enumeration value. The three types of return codes (informational, API error, and execution error) and their corresponding values are listed in Table 3.16.

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

(a) Field values of type GrB_Orientation.

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

Table 3.16: Enumeration literals and corresponding values returned by GraphBLAS methods and operations.

(a) Informational return values

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

(b) API errors

Symbol	Value	Description
GrB_UNINITIALIZED_OBJECT	-1	A GraphBLAS object is passed to a method before <code>new</code> was called on it.
GrB_NULL_POINTER	-2	A NULL is passed for a pointer parameter.
GrB_INVALID_VALUE	-3	Miscellaneous incorrect values.
GrB_INVALID_INDEX	-4	Indices passed are larger than dimensions of the matrix or vector being accessed.
GrB_DOMAIN_MISMATCH	-5	A mismatch between domains of collections and operations when user-defined domains are in use.
GrB_DIMENSION_MISMATCH	-6	Operations on matrices and vectors with incompatible dimensions.
GrB_OUTPUT_NOT_EMPTY	-7	An attempt was made to build a matrix or vector using an output object that already contains valid tuples (elements).
GrB_NOT_IMPLEMENTED	-8	An attempt was made to call a GraphBLAS method for a combination of input parameters that is not supported by a particular implementation.
GrB_ALREADY_SET	-9	An attempt was made to write to a field which may only be written to once.

(c) Execution errors

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

Chapter 4

Methods

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

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

4.1 Context methods

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

4.1.1 `init`: Initialize a GraphBLAS context

Creates and initializes a GraphBLAS C API context.

C Syntax

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

Parameters

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

1094 **Return Values**

1095 GrB_SUCCESS operation completed successfully.

1096 GrB_PANIC unknown internal error.

1097 GrB_INVALID_VALUE invalid mode specified, or method called multiple times.

1098 **Description**

1099 The init method creates and initializes a GraphBLAS C API context. The argument to GrB_init
1100 defines the mode for the context. The two available modes are:

- 1101 • GrB_BLOCKING: In this mode, each method in a sequence returns after its computations have
1102 completed and output arguments are available to subsequent statements in an application.
1103 When executing in GrB_BLOCKING mode, the methods execute in program order.
- 1104 • GrB_NONBLOCKING: In this mode, methods in a sequence may return after arguments in
1105 the method have been tested for dimension and domain compatibility within the method
1106 but potentially before their computations complete. Output arguments are available to sub-
1107 sequent GraphBLAS methods in an application. When executing in GrB_NONBLOCKING
1108 mode, the methods in a sequence may execute in any order that preserves the mathematical
1109 result defined by the sequence.

1110 An application can only create one context per execution instance. An application may only call
1111 GrB_Init once. Calling GrB_Init more than once results in undefined behavior.

1112 **4.1.2 finalize: Finalize a GraphBLAS context**

1113 Terminates and frees any internal resources created to support the GraphBLAS C API context.

1114 **C Syntax**

1115 GrB_Info GrB_finalize();

1116 **Return Values**

1117 GrB_SUCCESS operation completed successfully.

1118 GrB_PANIC unknown internal error.

1119 **Description**

1120 The `finalize` method terminates and frees any internal resources created to support the GraphBLAS
1121 C API context. `GrB_finalize` may only be called after a context has been initialized by calling
1122 `GrB_init`, or else undefined behavior occurs. After `GrB_finalize` has been called to finalize a Graph-
1123 BLAS context, calls to any GraphBLAS methods, including `GrB_finalize`, will result in undefined
1124 behavior.

1125 **4.1.3 getVersion: Get the version number of the standard.**

1126 Query the library for the version number of the standard that this library implements.

1127 **C Syntax**

```
1128         GrB_Info GrB_getVersion(unsigned int *version,  
1129                                unsigned int *subversion);
```

1130 **Parameters**

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

1132 version (OUT) On successful return will hold the value of the subversion number.

1133 **Return Values**

1134 GrB_SUCCESS operation completed successfully.

1135 GrB_PANIC unknown internal error.

1136 **Description**

1137 The `getVersion` method is used to query the major and minor version number of the GraphBLAS
1138 C API specification that the library implements at runtime. To support compile time queries the
1139 following two macros shall also be defined by the library.

```
1140         #define GRB_VERSION      2  
1141         #define GRB_SUBVERSION  0
```

1142 **4.2 Object methods**

1143 This section describes methods that setup and operate on GraphBLAS opaque objects but are not
1144 part of the the GraphBLAS math specification.

1145 4.2.1 Get and Set methods

1146 The methods in this section query and, optionally, set internal fields of GraphBLAS objects.

1147 4.2.1.1 get: Query the value of an object

1148 C Syntax

```
1149 GrB_Info GrB_get(GrB_<OBJ> o, <type> value, GrB_Field field);
```

1150 Parameters

1151 OBJ (IN) An existing, valid GraphBLAS object (collection, operation, type) which is
1152 being queried. To indicate the global context, the constant `GrB_Global` is used.

1153 value (OUT) A pointer to or `GrB_Scalar` containing a value whose type is dependent on
1154 field which will be filled with the current value of the field. type may be `int32_t*`,
1155 `size_t*`, `GrB_Scalar`, `char*` or `void*`.

1156 field (IN) The field being queried.

1157 Return Value

1158 `GrB_SUCCESS` The method completed successfully.

1159 `GrB_PANIC` unknown internal error.

1160 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1161 `GrB_UNINITIALIZED_OBJECT` the value parameter is `GrB_Scalar` and has not been initialized by
1162 a call to `new`.

1163 `GrB_INVALID_VALUE` invalid value type provided for the field or invalid field.

1164 Description

1165 Queries a field of an existing GraphBLAS object. The type of the argument is uniquely determined
1166 by field. For the case of `char*` and `void*`, the value can be replaced with `size_t*` to get the required
1167 buffer size to hold the response. Fields marked as hints in Table 3.13 will return a hint on how
1168 best to use the object.

1169 4.2.1.2 set: Set field of an object

1170 Set the content for a field for an existing GraphBLAS object.

1171 C Syntax

```
1172     GrB_Info GrB_set(GrB_<OBJ> o, <type> value, GrB_Field field);
1173     GrB_Info GrB_set(GrB_<OBJ> o, void *value, GrB_Field field, size_t voidSize);
```

1174 Parameters

1175 OBJ (IN) The GraphBLAS object which is having field set. To indicate
1176 the global context, the constant `GrB_Global` is used.

1177 value (IN) A value whose type is dependent on field. type may be a
1178 `int32_t`, `GrB_Scalar`, `char*` or `void*`.

1179 field (IN) The field being set.

1180 voidSize (IN) The size of the `void*` buffer. Note that a size is not needed for
1181 `char*` because the string is assumed null-terminated.

1182 Return Values

1183 `GrB_SUCCESS` The method completed successfully.

1184 `GrB_PANIC` unknown internal error.

1185 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1186 `GrB_UNINITIALIZED_OBJECT` the `GrB_Scalar` parameter has not been initialized by a call to `new`.

1187 `GrB_INVALID_VALUE` invalid value set on the field, invalid field, or field is read-only.

1188 `GrB_ALREADY_SET` this field has already been set, and may only be set once.

1189 Description

1190 Set a field of OBJ or the Global context to a new value.

1191 4.2.2 Algebra methods

1192 4.2.2.1 Type_new: Construct a new GraphBLAS (user-defined) type

1193 Creates a new user-defined GraphBLAS type. This type can then be used to create new operators,
1194 monoids, semirings, vectors and matrices.

1195 C Syntax

```
1196         GrB_Info GrB_Type_new(GrB_Type  *utype,  
1197                               size_t     sizeof(ctype));
```

1198 Parameters

1199 `utype` (INOUT) On successful return, contains a handle to the newly created user-defined
1200 GraphBLAS type object.

1201 `ctype` (IN) A C type that defines the new GraphBLAS user-defined type.

1202 Return Values

1203 `GrB_SUCCESS` operation completed successfully.

1204 `GrB_PANIC` unknown internal error.

1205 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1206 `GrB_NULL_POINTER` `utype` pointer is NULL.

1207 Description

1208 Given a C type `ctype`, the `Type_new` method returns in `utype` a handle to a new GraphBLAS type
1209 that is equivalent to the C type. Variables of this `ctype` must be a struct, union, or fixed-size array.
1210 In particular, given two variables, `src` and `dst`, of type `ctype`, the following operation must be a
1211 valid way to copy the contents of `src` to `dst`:

```
1212         memcpy(&dst, &src, sizeof(ctype))
```

1213 A new, user-defined type `utype` should be destroyed with a call to `GrB_free(utype)` when no longer
1214 needed.

1215 It is not an error to call this method more than once on the same variable; however, the handle to
1216 the previously created object will be overwritten.

1217 4.2.2.2 UnaryOp_new: Construct a new GraphBLAS unary operator

1218 Initializes a new GraphBLAS unary operator with a specified user-defined function and its types
1219 (domains).

1220 C Syntax

```
1221     GrB_Info GrB_UnaryOp_new(GrB_UnaryOp *unary_op,  
1222                             void          (*unary_func)(void*, const void*),  
1223                             GrB_Type      d_out,  
1224                             GrB_Type      d_in);
```

1225 Parameters

1226 unary_op (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1227 unary operator object.

1228 unary_func (IN) a pointer to a user-defined function that takes one input parameter of d_in's
1229 type and returns a value of d_out's type, both passed as void pointers. Specifically
1230 the signature of the function is expected to be of the form:

```
1231         void func(void *out, const void *in);
```

1233 d_out (IN) The GrB_Type of the return value of the unary operator being created. Should
1234 be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-
1235 BLAS type.

1236 d_in (IN) The GrB_Type of the input argument of the unary operator being created.
1237 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined
1238 GraphBLAS type.

1239 Return Values

1240 GrB_SUCCESS operation completed successfully.

1241 GrB_PANIC unknown internal error.

1242 GrB_OUT_OF_MEMORY not enough memory available for operation.

1243 GrB_UNINITIALIZED_OBJECT any GrB_Type parameter (for user-defined types) has not been ini-
1244 tialized by a call to GrB_Type_new.

1245 GrB_NULL_POINTER unary_op or unary_func pointers are NULL.

1246 Description

1247 The UnaryOp_new method creates a new GraphBLAS unary operator

1248 $f_u = \langle \mathbf{D}(d_out), \mathbf{D}(d_in), \text{unary_func} \rangle$

1249 and returns a handle to it in `unary_op`.

1250 The implementation of `unary_func` must be such that it works even if the `d_out` and `d_in` arguments
1251 are aliased. In other words, for all invocations of the function:

```
1252     unary_func(out,in);
```

1253 the value of `out` must be the same as if the following code was executed:

```
1254     D(d_in) *tmp = malloc(sizeof(D(d_in)));
1255     memcpy(tmp,in,sizeof(D(d_in)));
1256     unary_func(out,tmp);
1257     free(tmp);
```

1258 It is not an error to call this method more than once on the same variable; however, the handle to
1259 the previously created object will be overwritten.

1260 4.2.2.3 BinaryOp_new: Construct a new GraphBLAS binary operator

1261 Initializes a new GraphBLAS binary operator with a specified user-defined function and its types
1262 (domains).

1263 C Syntax

```
1264     GrB_Info GrB_BinaryOp_new(GrB_BinaryOp *binary_op,
1265                               void          (*binary_func)(void*,
1266                               const void*,
1267                               const void*),
1268                               GrB_Type      d_out,
1269                               GrB_Type      d_in1,
1270                               GrB_Type      d_in2);
```

1271 Parameters

1272 `binary_op` (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1273 binary operator object.

1274 `binary_func` (IN) A pointer to a user-defined function that takes two input parameters of types
1275 `d_in1` and `d_in2` and returns a value of type `d_out`, all passed as void pointers.
1276 Specifically the signature of the function is expected to be of the form:

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

1278

1279 **d_out** (IN) The **GrB_Type** of the return value of the binary operator being created. Should
1280 be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-
1281 BLAS type.

1282 **d_in1** (IN) The **GrB_Type** of the left hand argument of the binary operator being created.
1283 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined
1284 GraphBLAS type.

1285 **d_in2** (IN) The **GrB_Type** of the right hand argument of the binary operator being cre-
1286 ated. Should be one of the predefined GraphBLAS types in Table 3.2, or a user-
1287 defined GraphBLAS type.

1288 **Return Values**

1289 **GrB_SUCCESS** operation completed successfully.

1290 **GrB_PANIC** unknown internal error.

1291 **GrB_OUT_OF_MEMORY** not enough memory available for operation.

1292 **GrB_UNINITIALIZED_OBJECT** the **GrB_Type** (for user-defined types) has not been initialized by a
1293 call to **GrB_Type_new**.

1294 **GrB_NULL_POINTER** **binary_op** or **binary_func** pointer is **NULL**.

1295 **Description**

1296 The **BinaryOp_new** methods creates a new GraphBLAS binary operator

1297 $f_b = \langle \mathbf{D}(\mathbf{d_out}), \mathbf{D}(\mathbf{d_in1}), \mathbf{D}(\mathbf{d_in2}), \mathbf{binary_func} \rangle$

1298 and returns a handle to it in **binary_op**.

1299 The implementation of **binary_func** must be such that it works even if any of the **d_out**, **d_in1**, and
1300 **d_in2** arguments are aliased to each other. In other words, for all invocations of the function:

1301 **binary_func(out, in1, in2);**

1302 the value of **out** must be the same as if the following code was executed:

```
1303        D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));
1304        D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));
1305        memcpy(tmp1, in1, sizeof(D(d_in1)));
1306        memcpy(tmp2, in2, sizeof(D(d_in2)));
1307        binary_func(out, tmp1, tmp2);
1308        free(tmp2);
1309        free(tmp1);
```

1310 It is not an error to call this method more than once on the same variable; however, the handle to
1311 the previously created object will be overwritten.

1312 4.2.2.4 Monoid_new: Construct a new GraphBLAS monoid

1313 Creates a new monoid with specified binary operator and identity value.

1314 C Syntax

```
1315         GrB_Info GrB_Monoid_new(GrB_Monoid    *monoid,  
1316                               GrB_BinaryOp    binary_op,  
1317                               <type>          identity);
```

1318 Parameters

1319 monoid (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1320 monoid object.

1321 binary_op (IN) An existing GraphBLAS associative binary operator whose input and output
1322 types are the same.

1323 identity (IN) The value of the identity element of the monoid. Must be the same type as
1324 the type used by the **binary_op** operator.

1325 Return Values

1326 GrB_SUCCESS operation completed successfully.

1327 GrB_PANIC unknown internal error.

1328 GrB_OUT_OF_MEMORY not enough memory available for operation.

1329 GrB_UNINITIALIZED_OBJECT the GrB_BinaryOp (for user-defined operators) has not been initial-
1330 ized by a call to GrB_BinaryOp_new.

1331 GrB_NULL_POINTER monoid pointer is NULL.

1332 GrB_DOMAIN_MISMATCH all three argument types of the binary operator and the type of the
1333 identity value are not the same.

1334 Description

1335 The **Monoid_new** method creates a new monoid $M = \langle \mathbf{D}(\text{binary_op}), \text{binary_op}, \text{identity} \rangle$ and re-
1336 turns a handle to it in **monoid**.

1337 If `binary_op` is not associative, the results of GraphBLAS operations that require associativity of
1338 this monoid will be undefined.

1339 It is not an error to call this method more than once on the same variable; however, the handle to
1340 the previously created object will be overwritten.

1341 4.2.2.5 Semiring_new: Construct a new GraphBLAS semiring

1342 Creates a new semiring with specified domain, operators, and elements.

1343 C Syntax

```
1344         GrB_Info GrB_Semiring_new(GrB_Semiring *semiring,  
1345                                   GrB_Monoid    add_op,  
1346                                   GrB_BinaryOp   mul_op);
```

1347 Parameters

1348 `semiring` (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1349 `semiring`.

1350 `add_op` (IN) An existing GraphBLAS commutative monoid that specifies the addition op-
1351 erator and its identity.

1352 `mul_op` (IN) An existing GraphBLAS binary operator that specifies the semiring's multi-
1353 plication operator. In addition, `mul_op`'s output domain, $\mathbf{D}_{out}(\text{mul_op})$, must be
1354 the same as the `add_op`'s domain $\mathbf{D}(\text{add_op})$.

1355 Return Values

1356 `GrB_SUCCESS` operation completed successfully.

1357 `GrB_PANIC` unknown internal error.

1358 `GrB_OUT_OF_MEMORY` not enough memory available for this method to complete.

1359 `GrB_UNINITIALIZED_OBJECT` the `add_op` (for user-define monoids) object has not been initialized
1360 with a call to `GrB_Monoid_new` or the `mul_op` (for user-defined
1361 operators) object has not been not been initialized by a call to
1362 `GrB_BinaryOp_new`.

1363 `GrB_NULL_POINTER` `semiring` pointer is NULL.

1364 `GrB_DOMAIN_MISMATCH` the output domain of `mul_op` does not match the domain of the
1365 `add_op` monoid.

Description

The `Semiring_new` method creates a new semiring:

$$S = \langle \mathbf{D}_{out}(\text{mul_op}), \mathbf{D}_{in_1}(\text{mul_op}), \mathbf{D}_{in_2}(\text{mul_op}), \text{add_op}, \text{mul_op}, \mathbf{0}(\text{add_op}) \rangle$$

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

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

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

4.2.2.6 IndexUnaryOp_new: Construct a new GraphBLAS index unary operator [Scott: NEW CONTENT]

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

C Syntax

```
GrB_Info GrB_IndexUnaryOp_new(GrB_IndexUnaryOp *index_unary_op,
                              void (*index_unary_func)(void*,
                                                         const void*,
                                                         GrB_Index,
                                                         GrB_Index,
                                                         const void*),
                              GrB_Type          d_out,
                              GrB_Type          d_in1,
                              GrB_Type          d_in2);
```

Parameters

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

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

```
void func(void *out,
          const void *in1,
          GrB_Index row_index,
          GrB_Index col_index,
```

1398 `const void *in2);`
1399

1400 **d_out** (IN) The `GrB_Type` of the return value of the index unary operator being created.
1401 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined
1402 GraphBLAS type.

1403 **d_in1** (IN) The `GrB_Type` of the first input argument of the index unary operator being
1404 created and corresponds to the stored values of the `GrB_Vector` or `GrB_Matrix`
1405 being operated on. Should be one of the predefined GraphBLAS types in Ta-
1406 ble 3.2, or a user-defined GraphBLAS type.

1407 **d_in2** (IN) The `GrB_Type` of the last input argument of the index unary operator be-
1408 ing created and corresponds to a scalar provided by the GraphBLAS operation
1409 that uses this operator. Should be one of the predefined GraphBLAS types in
1410 Table 3.2, or a user-defined GraphBLAS type.

1411 Return Values

1412 `GrB_SUCCESS` operation completed successfully.

1413 `GrB_PANIC` unknown internal error.

1414 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1415 `GrB_UNINITIALIZED_OBJECT` the `GrB_Type` (for user-defined types) has not been initialized by a
1416 call to `GrB_Type_new`.

1417 `GrB_NULL_POINTER` `index_unary_op` or `index_unary_func` pointer is `NULL`.

1418 Description

1419 The `IndexUnaryOp_new` methods creates a new GraphBLAS index unary operator

1420
$$f_i = \langle \mathbf{D}(\mathbf{d_out}), \mathbf{D}(\mathbf{d_in1}), \mathbf{D}(\mathbf{GrB_Index}), \mathbf{D}(\mathbf{GrB_Index}), \mathbf{D}(\mathbf{d_in2}), \text{index_unary_func} \rangle$$

1421 and returns a handle to it in `index_unary_op`.

1422 The implementation of `index_unary_func` must be such that it works even if any of the `d_out`,
1423 `d_in1`, and `d_in2` arguments are aliased to each other. In other words, for all invocations of the
1424 function:

1425 `index_unary_func(out, in1, row_index, col_index, n, in2);`

1426 the value of `out` must be the same as if the following code was executed (shown here for matrices):

```

1427     GrB_Index row_index = ...;
1428     GrB_Index col_index = ...;
1429     D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));
1430     D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));
1431     memcpy(tmp1,in1,sizeof(D(d_in1)));
1432     memcpy(tmp2,in2,sizeof(D(d_in2)));
1433     index_unary_func(out,tmp1,row_index,col_index,tmp2);
1434     free(tmp2);
1435     free(tmp1);

```

1436 It is not an error to call this method more than once on the same variable; however, the handle to
1437 the previously created object will be overwritten.

1438 4.2.3 Scalar methods

1439 4.2.3.1 Scalar_new: Construct a new scalar

1440 Creates a new empty scalar with specified domain.

1441 C Syntax

```

1442     GrB_Info GrB_Scalar_new(GrB_Scalar *s,
1443                             GrB_Type    d);

```

1444 Parameters

1445 **s** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1446 scalar.

1447 **d** (IN) The type corresponding to the domain of the scalar being created. Can be
1448 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined
1449 GraphBLAS type.

1450 Return Values

1451 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1452 blocking mode, this indicates that the API checks for the input
1453 arguments passed successfully. Either way, output scalar **s** is ready
1454 to be used in the next method of the sequence.

1455 **GrB_PANIC** Unknown internal error.

1456 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1457 GraphBLAS objects (input or output) is in an invalid state caused

1458 by a previous execution error. Call `GrB_error()` to access any error
1459 messages generated by the implementation.

1460 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1461 **GrB_UNINITIALIZED_OBJECT** The `GrB_Type` object has not been initialized by a call to `GrB_Type_new`
1462 (needed for user-defined types).

1463 **GrB_NULL_POINTER** The `s` pointer is `NULL`.

1464 Description

1465 Creates a new GraphBLAS scalar s of domain $\mathbf{D}(d)$ and empty $\mathbf{L}(s)$. The method returns a handle
1466 to the new scalar in `s`.

1467 It is not an error to call this method more than once on the same variable; however, the handle to
1468 the previously created object will be overwritten.

1469 4.2.3.2 Scalar_dup: Construct a copy of a GraphBLAS scalar

1470 Creates a new scalar with the same domain and contents as another scalar.

1471 C Syntax

```
1472 GrB_Info GrB_Scalar_dup(GrB_Scalar      *t,  
1473                          const GrB_Scalar s);
```

1474 Parameters

1475 `t` (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1476 scalar.

1477 `s` (IN) The GraphBLAS scalar to be duplicated.

1478 Return Values

1479 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1480 blocking mode, this indicates that the API checks for the input
1481 arguments passed successfully. Either way, output scalar `t` is ready
1482 to be used in the next method of the sequence.

1483 **GrB_PANIC** Unknown internal error.

1484 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1485 GraphBLAS objects (input or output) is in an invalid state caused

1486 by a previous execution error. Call `GrB_error()` to access any error
 1487 messages generated by the implementation.

1488 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1489 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS scalar, `s`, has not been initialized by a call to
 1490 `Scalar_new` or `Scalar_dup`.

1491 `GrB_NULL_POINTER` The `t` pointer is NULL.

1492 Description

1493 Creates a new scalar t of domain $\mathbf{D}(\mathbf{s})$ and contents $\mathbf{L}(\mathbf{s})$. The method returns a handle to the new
 1494 scalar in `t`.

1495 It is not an error to call this method more than once with the same output variable; however, the
 1496 handle to the previously created object will be overwritten.

1497 4.2.3.3 `Scalar_clear`: Clear/remove a stored value from a scalar

1498 Removes the stored value from a scalar.

1499 C Syntax

1500 `GrB_Info GrB_Scalar_clear(GrB_Scalar s);`

1501 Parameters

1502 `s` (INOUT) An existing GraphBLAS scalar to clear.

1503 Return Values

1504 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
 1505 blocking mode, this indicates that the API checks for the input
 1506 arguments passed successfully. Either way, output scalar `s` is ready
 1507 to be used in the next method of the sequence.

1508 `GrB_PANIC` Unknown internal error.

1509 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
 1510 GraphBLAS objects (input or output) is in an invalid state caused
 1511 by a previous execution error. Call `GrB_error()` to access any error
 1512 messages generated by the implementation.

1513 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1514 GrB_UNINITIALIZED_OBJECT The GraphBLAS scalar, *s*, has not been initialized by a call to
1515 Scalar_new or Scalar_dup.

1516 Description

1517 Removes the stored value from an existing scalar. After the call, *L(s)* is empty. The size of the
1518 scalar does not change.

1519 4.2.3.4 Scalar_nvals: Number of stored elements in a scalar

1520 Retrieve the number of stored elements in a scalar (either zero or one).

1521 C Syntax

```
1522         GrB_Info GrB_Scalar_nvals(GrB_Index      *nvals,  
1523                                   const GrB_Scalar s);
```

1524 Parameters

1525 *nvals* (OUT) On successful return, this is set to the number of stored elements in the
1526 scalar (zero or one).

1527 *s* (IN) An existing GraphBLAS scalar being queried.

1528 Return Values

1529 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1530 cessfully and the value of *nvals* has been set.

1531 GrB_PANIC Unknown internal error.

1532 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1533 GraphBLAS objects (input or output) is in an invalid state caused
1534 by a previous execution error. Call GrB_error() to access any error
1535 messages generated by the implementation.

1536 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1537 GrB_UNINITIALIZED_OBJECT The GraphBLAS scalar, *s*, has not been initialized by a call to
1538 Scalar_new or Scalar_dup.

1539 GrB_NULL_POINTER The *nvals* pointer is NULL.

1540 Description

1541 Return **nvals(s)** in **nvals**. This is the number of stored elements in scalar **s**, which is the size of
1542 **L(s)**, and can only be either zero or one (see Section 3.5.1).

1543 4.2.3.5 Scalar_setElement: Set the single element in a scalar

1544 Set the single element of a scalar to a given value.

1545 C Syntax

```
1546      GrB_Info GrB_Scalar_setElement(GrB_Scalar    s,  
1547                                   <type>        val);
```

1548 Parameters

1549 **s** (INOUT) An existing GraphBLAS scalar for which the element is to be assigned.

1550 **val** (IN) Scalar value to assign. The type must be compatible with the domain of **s**.

1551 Return Values

1552 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1553 blocking mode, this indicates that the compatibility tests on in-
1554 dex/dimensions and domains for the input arguments passed suc-
1555 cessfully. Either way, the output scalar **s** is ready to be used in the
1556 next method of the sequence.

1557 **GrB_PANIC** Unknown internal error.

1558 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1559 GraphBLAS objects (input or output) is in an invalid state caused
1560 by a previous execution error. Call **GrB_error()** to access any error
1561 messages generated by the implementation.

1562 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1563 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS scalar, **s**, has not been initialized by a call to
1564 **Scalar_new** or **Scalar_dup**.

1565 **GrB_DOMAIN_MISMATCH** The domains of **s** and **val** are incompatible.

Description

First, `val` and output GraphBLAS scalar are tested for domain compatibility as follows: $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}(\mathbf{s})$. 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_Scalar_setElement` ends and the domain mismatch error listed above is returned.

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

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

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

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

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

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

C Syntax

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

Parameters

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

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

Return Values

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

`GrB_PANIC` Unknown internal error.

1627 Parameters

- 1628 **v** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1629 vector.
- 1630 **d** (IN) The type corresponding to the domain of the vector being created. Can be
1631 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined
1632 GraphBLAS type.
- 1633 **nsz** (IN) The size of the vector being created.

1634 Return Values

- 1635 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1636 blocking mode, this indicates that the API checks for the input
1637 arguments passed successfully. Either way, output vector **v** is ready
1638 to be used in the next method of the sequence.
- 1639 **GrB_PANIC** Unknown internal error.
- 1640 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1641 GraphBLAS objects (input or output) is in an invalid state caused
1642 by a previous execution error. Call **GrB_error()** to access any error
1643 messages generated by the implementation.
- 1644 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.
- 1645 **GrB_UNINITIALIZED_OBJECT** The **GrB_Type** object has not been initialized by a call to **GrB_Type_new**
1646 (needed for user-defined types).
- 1647 **GrB_NULL_POINTER** The **v** pointer is **NULL**.
- 1648 **GrB_INVALID_VALUE** **nsz** is zero or outside the range of the type **GrB_Index**.

1649 Description

- 1650 Creates a new vector **v** of domain **D(d)**, size **nsz**, and empty **L(v)**. The method returns a handle
1651 to the new vector in **v**.
- 1652 It is not an error to call this method more than once on the same variable; however, the handle to
1653 the previously created object will be overwritten.

1654 4.2.4.2 Vector_dup: Construct a copy of a GraphBLAS vector

- 1655 Creates a new vector with the same domain, size, and contents as another vector.

1656 C Syntax

```
1657         GrB_Info GrB_Vector_dup(GrB_Vector      *w,  
1658                                const GrB_Vector  u);
```

1659 Parameters

1660 **w** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1661 vector.

1662 **u** (IN) The GraphBLAS vector to be duplicated.

1663 Return Values

1664 **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 **w** is ready
1667 to be used in the next method of the sequence.

1668 **GrB_PANIC** Unknown internal error.

1669 **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 **GrB_error()** to access any error
1672 messages generated by the implementation.

1673 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1674 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS vector, **u**, has not been initialized by a call to
1675 **Vector_new** or **Vector_dup**.

1676 **GrB_NULL_POINTER** The **w** pointer is **NULL**.

1677 Description

1678 Creates a new vector **w** of domain **D(u)**, size **size(u)**, and contents **L(u)**. The method returns a
1679 handle to the new vector in **w**.

1680 It is not an error to call this method more than once on the same variable; however, the handle to
1681 the previously created object will be overwritten.

1682 4.2.4.3 Vector_resize: Resize a vector

1683 Changes the size of an existing vector.

1684 C Syntax

```
1685         GrB_Info GrB_Vector_resize(GrB_Vector  w,  
1686                                   GrB_Index   nsize);
```

1687 Parameters

1688 **w** (INOUT) An existing Vector object that is being resized.

1689 **nsize** (IN) The new size of the vector. It can be smaller or larger than the current size.

1690 Return Values

1691 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1692 blocking mode, this indicates that the API checks for the input
1693 arguments passed successfully. Either way, output vector **w** is ready
1694 to be used in the next method of the sequence.

1695 **GrB_PANIC** Unknown internal error.

1696 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1697 GraphBLAS objects (input or output) is in an invalid state caused
1698 by a previous execution error. Call **GrB_error()** to access any error
1699 messages generated by the implementation.

1700 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1701 **GrB_NULL_POINTER** The **w** pointer is **NULL**.

1702 **GrB_INVALID_VALUE** **nsize** is zero or outside the range of the type **GrB_Index**.

1703 Description

1704 Changes the size of **w** to **nsize**. The domain **D(w)** of vector **w** remains the same. The contents **L(w)**
1705 are modified as described below.

1706 Let $w = \langle \mathbf{D}(w), N, \mathbf{L}(w) \rangle$ when the method is called. When the method returns, $w = \langle \mathbf{D}(w), \mathbf{nsize}, \mathbf{L}'(w) \rangle$
1707 where $\mathbf{L}'(w) = \{(i, w_i) : (i, w_i) \in \mathbf{L}(w) \wedge (i < \mathbf{nsize})\}$. That is, all elements of **w** with index greater
1708 than or equal to the new vector size (**nsize**) are dropped.

1709 4.2.4.4 Vector_clear: Clear a vector

1710 Removes all the elements (tuples) from a vector.

1711 C Syntax

1712 GrB_Info GrB_Vector_clear(GrB_Vector v);

1713 Parameters

1714 v (INOUT) An existing GraphBLAS vector to clear.

1715 Return Values

1716 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1717 blocking mode, this indicates that the API checks for the input
1718 arguments passed successfully. Either way, output vector v is ready
1719 to be used in the next method of the sequence.

1720 GrB_PANIC Unknown internal error.

1721 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1722 GraphBLAS objects (input or output) is in an invalid state caused
1723 by a previous execution error. Call GrB_error() to access any error
1724 messages generated by the implementation.

1725 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1726 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, v, has not been initialized by a call to
1727 Vector_new or Vector_dup.

1728 Description

1729 Removes all elements (tuples) from an existing vector. After the call to GrB_Vector_clear(v),
1730 $L(v) = \emptyset$. The size of the vector does not change.

1731 4.2.4.5 Vector_size: Size of a vector

1732 Retrieve the size of a vector.

1733 C Syntax

1734 GrB_Info GrB_Vector_size(GrB_Index *nsize,
1735 const GrB_Vector v);

1736 Parameters

1737 **nsiz** (OUT) On successful return, is set to the size of the vector.

1738 **v** (IN) An existing GraphBLAS vector being queried.

1739 Return Values

1740 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
1741 cessfully and the value of **nsiz** has been set.

1742 **GrB_PANIC** Unknown internal error.

1743 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1744 GraphBLAS objects (input or output) is in an invalid state caused
1745 by a previous execution error. Call **GrB_error()** to access any error
1746 messages generated by the implementation.

1747 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS vector, **v**, has not been initialized by a call to
1748 **Vector_new** or **Vector_dup**.

1749 **GrB_NULL_POINTER** **nsiz** pointer is **NULL**.

1750 Description

1751 Return **size(v)** in **nsiz**.

1752 4.2.4.6 Vector_nvals: Number of stored elements in a vector

1753 Retrieve the number of stored elements (tuples) in a vector.

1754 C Syntax

```
1755            GrB_Info GrB_Vector_nvals(GrB_Index            *nvals,  
1756                                        const GrB_Vector    v);
```

1757 Parameters

1758 **nvals** (OUT) On successful return, this is set to the number of stored elements (tuples)
1759 in the vector.

1760 **v** (IN) An existing GraphBLAS vector being queried.

1761 Return Values

1762 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1763 cessfully and the value of `nvals` has been set.

1764 GrB_PANIC Unknown internal error.

1765 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1766 GraphBLAS objects (input or output) is in an invalid state caused
1767 by a previous execution error. Call `GrB_error()` to access any error
1768 messages generated by the implementation.

1769 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1770 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, `v`, has not been initialized by a call to
1771 Vector_new or Vector_dup.

1772 GrB_NULL_POINTER The `nvals` pointer is NULL.

1773 Description

1774 Return `nvals(v)` in `nvals`. This is the number of stored elements in vector `v`, which is the size of
1775 `L(v)` (see Section 3.5.2).

1776 4.2.4.7 Vector_build: Store elements from tuples into a vector

1777 C Syntax

```
1778                   GrB_Info GrB_Vector_build(GrB_Vector                   w,  
1779                                           const GrB_Index               *indices,  
1780                                           const <type>                *values,  
1781                                           GrB_Index                    n,  
1782                                           const GrB_BinaryOp           dup);
```

1783 Parameters

1784 w (INOUT) An existing Vector object to store the result.

1785 indices (IN) Pointer to an array of indices.

1786 values (IN) Pointer to an array of scalars of a type that is compatible with the domain of
1787 vector `w`.

1788 n (IN) The number of entries contained in each array (the same for `indices` and `values`).

1789 **dup** (IN) An associative and commutative binary operator to apply when duplicate
 1790 values for the same location are present in the input arrays. All three domains of
 1791 **dup** must be the same; hence $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$. If **dup** is **GrB_NULL**,
 1792 then duplicate locations will result in an error.

1793 Return Values

1794 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 1795 blocking mode, this indicates that the API checks for the input
 1796 arguments passed successfully. Either way, output vector **w** is
 1797 ready to be used in the next method of the sequence.

1798 **GrB_PANIC** Unknown internal error.

1799 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
 1800 opaque GraphBLAS objects (input or output) is in an invalid
 1801 state caused by a previous execution error. Call **GrB_error()** to
 1802 access any error messages generated by the implementation.

1803 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1804 **GrB_UNINITIALIZED_OBJECT** Either **w** has not been initialized by a call to **GrB_Vector_new**
 1805 or by **GrB_Vector_dup**, or **dup** has not been initialized by a call
 1806 to **GrB_BinaryOp_new**.

1807 **GrB_NULL_POINTER** indices or values pointer is **NULL**.

1808 **GrB_INDEX_OUT_OF_BOUNDS** A value in indices is outside the allowed range for **w**.

1809 **GrB_DOMAIN_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are
 1810 not all the same, or the domains of **values** and **w** are incompatible
 1811 with each other or D_{dup} .

1812 **GrB_OUTPUT_NOT_EMPTY** Output vector **w** already contains valid tuples (elements). In
 1813 other words, **GrB_Vector_nvals(C)** returns a positive value.

1814 **GrB_INVALID_VALUE** indices contains a duplicate location and **dup** is **GrB_NULL**.

1815 Description

1816 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
 1817 from **w** in its domain; otherwise, $\tilde{\mathbf{w}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \emptyset \rangle$.

1818 Each tuple $\{\text{indices}[k], \text{values}[k]\}$, where $0 \leq k < n$, is a contribution to the output in the form of

$$1819 \quad \tilde{\mathbf{w}}(\text{indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \mathbf{dup} \neq \mathbf{GrB_NULL} \\ (\mathbf{D}(\mathbf{w})) \text{values}[k] & \text{otherwise.} \end{cases}$$

1820 If multiple values for the same location are present in the input arrays and `dup` is not `GrB_NULL`,
 1821 `dup` is used to reduce the values before assignment into $\tilde{\mathbf{w}}$ as follows:

$$1822 \quad \tilde{\mathbf{w}}_i = \bigoplus_{k: \text{indices}[k]=i} (D_{dup}) \text{values}[k],$$

1823 where \oplus is the `dup` binary operator. Finally, the resulting $\tilde{\mathbf{w}}$ is copied into `w` via typecasting its
 1824 values to `D(w)` if necessary. If \oplus is not associative or not commutative, the result is undefined.

1825 The nonopaque input arrays, `indices` and `values`, must be at least as large as `n`.

1826 It is an error to call this function on an output object with existing elements. In other words,
 1827 `GrB_Vector_nvals(w)` should evaluate to zero prior to calling this function.

1828 After `GrB_Vector_build` returns, it is safe for a programmer to modify or delete the arrays `indices`
 1829 or `values`.

1830 4.2.4.8 Vector_setElement: Set a single element in a vector

1831 Set one element of a vector to a given value.

1832 C Syntax

```
1833 // scalar value
1834 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1835                               <type>         val,
1836                               GrB_Index       index);
1837
1838 // GraphBLAS scalar
1839 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1840                               const GrB_Scalar s,
1841                               GrB_Index       index);
```

1842 Parameters

1843 `w` (INOUT) An existing GraphBLAS vector for which an element is to be assigned.

1844 `val` or `s` (IN) Scalar assign. Its domain (type) must be compatible with the domain of `w`.

1845 `index` (IN) The location of the element to be assigned.

1846 Return Values

1847 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
 1848 blocking mode, this indicates that the compatibility tests on in-
 1849 dex/dimensions and domains for the input arguments passed suc-

cessfully. Either way, the 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 The GraphBLAS vector, **w**, or GraphBLAS scalar, **s**, has not been initialized by a call to a respective constructor.

GrB_INVALID_INDEX index specifies a location that is outside the dimensions of **w**.

GrB_DOMAIN_MISMATCH The domains of the vector and the scalar are incompatible.

Description

First, the scalar and output vector are tested for domain compatibility as follows: **D(val)** or **D(s)** must be compatible with **D(w)**. 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_Vector_setElement** ends and the domain mismatch error listed above is returned.

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

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

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

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

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

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

In **GrB_BLOCKING** mode, the method exits with return value **GrB_SUCCESS** and the new contents of **w** is as defined above and fully computed. In **GrB_NONBLOCKING** mode, the method exits with return value **GrB_SUCCESS** and the new contents 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.

1883 4.2.4.9 Vector_removeElement: Remove an element from a vector

1884 Remove (annihilate) one stored element from a vector.

1885 C Syntax

```
1886         GrB_Info GrB_Vector_removeElement(GrB_Vector  w,  
1887                                           GrB_Index    index);
```

1888 Parameters

1889 w (INOUT) An existing GraphBLAS vector from which an element is to be removed.

1890 index (IN) The location of the element to be removed.

1891 Return Values

1892 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1893 blocking mode, this indicates that the compatibility tests on in-
1894 dex/dimensions and domains for the input arguments passed suc-
1895 cessfully. Either way, the output vector w is ready to be used in
1896 the next method of the sequence.

1897 GrB_PANIC Unknown internal error.

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

1902 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1903 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, w, has not been initialized by a call to
1904 Vector_new or Vector_dup.

1905 GrB_INVALID_INDEX index specifies a location that is outside the dimensions of w.

1906 Description

1907 First, the index parameter is checked for a valid value where the following condition must hold:

$$1908 \quad 0 \leq \text{index} < \text{size}(w)$$

1909 If this condition is violated, execution of GrB_Vector_removeElement ends and the invalid index
1910 error listed above is returned.

1911 We are now ready to carry out the removal of a value that may be stored at the location specified
 1912 by `index`. If a value does not exist at the specified location in `w`, no error is reported and the
 1913 operation has no effect on the state of `w`. In either case, the following will be true on return from
 1914 the method: `index ∉ ind(w)`.

1915 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents
 1916 of `w` is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with
 1917 return value `GrB_SUCCESS` and the new content of vector `w` is as defined above but may not be
 1918 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1919 4.2.4.10 `Vector_extractElement`: Extract a single element from a vector.

1920 Extract one element of a vector into a scalar.

1921 C Syntax

```
1922 // scalar value
1923 GrB_Info GrB_Vector_extractElement(<type>          *val,
1924                                   const GrB_Vector u,
1925                                   GrB_Index         index);
1926
1927 // GraphBLAS scalar
1928 GrB_Info GrB_Vector_extractElement(GrB_Scalar      s,
1929                                   const GrB_Vector u,
1930                                   GrB_Index         index);
```

1931 Parameters

1932 `val` or `s` (INOUT) An existing scalar of whose domain is compatible with the domain of vector
 1933 `u`. On successful return, this scalar holds the result of the extract. Any previous
 1934 value stored in `val` or `s` is overwritten.

1935 `u` (IN) The GraphBLAS vector from which an element is extracted.

1936 `index` (IN) The location in `u` to extract.

1937 Return Values

1938 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-
 1939 cessfully. This indicates that the compatibility tests on dimensions
 1940 and domains for the input arguments passed successfully, and the
 1941 output scalar, `val` or `s`, has been computed and is ready to be used
 1942 in the next method of the sequence.

1943 GrB_NO_VALUE When using the transparent scalar, `val`, this is returned when there
1944 is no stored value at specified location.

1945 GrB_PANIC Unknown internal error.

1946 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1947 GraphBLAS objects (input or output) is in an invalid state caused
1948 by a previous execution error. Call `GrB_error()` to access any error
1949 messages generated by the implementation.

1950 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1951 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, `u`, or scalar, `s`, has not been initialized by
1952 a call to a corresponding constructor.

1953 GrB_NULL_POINTER `val` pointer is NULL.

1954 GrB_INVALID_INDEX `index` specifies a location that is outside the dimensions of `w`.

1955 GrB_DOMAIN_MISMATCH The domains of the vector and scalar are incompatible.

1956 Description

1957 First, the scalar and input vector are tested for domain compatibility as follows: $\mathbf{D}(\text{val})$ or $\mathbf{D}(\mathbf{s})$
1958 must be compatible with $\mathbf{D}(\mathbf{u})$. Two domains are compatible with each other if values from
1959 one domain can be cast to values in the other domain as per the rules of the C language. In
1960 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
1961 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
1962 `GrB_Vector_extractElement` ends and the domain mismatch error listed above is returned.

1963 Then, the `index` parameter is checked for a valid value where the following condition must hold:

$$1964 \qquad 0 \leq \text{index} < \text{size}(\mathbf{u})$$

1965 If this condition is violated, execution of `GrB_Vector_extractElement` ends and the invalid index
1966 error listed above is returned.

1967 We are now ready to carry out the extract into the output scalar; that is:

$$1968 \qquad \left. \begin{array}{l} \mathbf{L}(\mathbf{s}) \\ \text{val} \end{array} \right\} = \mathbf{u}(\text{index})$$

1969 If $\text{index} \in \mathbf{ind}(\mathbf{u})$, then the corresponding value from `u` is copied into `s` or `val` with casting as
1970 necessary. If $\text{index} \notin \mathbf{ind}(\mathbf{u})$, then one of the follow occurs depending on output scalar type:

- 1971 • The GraphBLAS scalar, `s`, is cleared and `GrB_SUCCESS` is returned.
- 1972 • The non-opaque scalar, `val`, is unchanged, and `GrB_NO_VALUE` is returned.

1973 When using the non-opaque scalar variant (`val`) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`
 1974 mode, the new contents of `val` are as defined above if the method exits with return value `GrB_SUCCESS`
 1975 or `GrB_NO_VALUE`.

1976 When using the GraphBLAS scalar variant (`s`) with a `GrB_SUCCESS` return value, the method
 1977 exits and the new contents of `s` is as defined above and fully computed in `GrB_BLOCKING` mode.
 1978 In `GrB_NONBLOCKING` mode, the new contents of `s` is as defined above but may not be fully
 1979 computed; however, it can be used in the next GraphBLAS method call in a sequence.

1980 4.2.4.11 `Vector_extractTuples`: Extract tuples from a vector

1981 Extract the contents of a GraphBLAS vector into non-opaque data structures.

1982 C Syntax

```
1983      GrB_Info GrB_Vector_extractTuples(GrB_Index      *indices,
1984                                     <type>          *values,
1985                                     GrB_Index        *n,
1986                                     const GrB_Vector v);
1987
```

1988 `indices` (OUT) Pointer to an array of indices that is large enough to hold all of the stored
 1989 values' indices.

1990 `values` (OUT) Pointer to an array of scalars of a type that is large enough to hold all of
 1991 the stored values whose type is compatible with `D(v)`.

1992 `n` (INOUT) Pointer to a value indicating (on input) the number of elements the
 1993 `values` and `indices` arrays can hold. Upon return, it will contain the number of
 1994 values written to the arrays.

1995 `v` (IN) An existing GraphBLAS vector.

1996 Return Values

1997 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-
 1998 cessfully. This indicates that the compatibility tests on the input
 1999 argument passed successfully, and the output arrays, `indices` and
 2000 `values`, have been computed.

2001 `GrB_PANIC` Unknown internal error.

2002 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
 2003 GraphBLAS objects (input or output) is in an invalid state caused
 2004 by a previous execution error. Call `GrB_error()` to access any error
 2005 messages generated by the implementation.


```

2035         GrB_Index    nrows,
2036         GrB_Index    ncols);

```

2037 Parameters

2038 **A** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
2039 matrix.

2040 **d** (IN) The type corresponding to the domain of the matrix being created. Can be
2041 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined
2042 GraphBLAS type.

2043 **nrows** (IN) The number of rows of the matrix being created.

2044 **ncols** (IN) The number of columns of the matrix being created.

2045 Return Values

2046 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
2047 blocking mode, this indicates that the API checks for the input ar-
2048 guments passed successfully. Either way, output matrix **A** is ready
2049 to be used in the next method of the sequence.

2050 **GrB_PANIC** Unknown internal error.

2051 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2052 GraphBLAS objects (input or output) is in an invalid state caused
2053 by a previous execution error. Call **GrB_error()** to access any error
2054 messages generated by the implementation.

2055 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2056 **GrB_UNINITIALIZED_OBJECT** The **GrB_Type** object has not been initialized by a call to **GrB_Type_new**
2057 (needed for user-defined types).

2058 **GrB_NULL_POINTER** The **A** pointer is **NULL**.

2059 **GrB_INVALID_VALUE** **nrows** or **ncols** is zero or outside the range of the type **GrB_Index**.

2060 Description

2061 Creates a new matrix **A** of domain **D(d)**, size **nrows** \times **ncols**, and empty **L(A)**. The method returns
2062 a handle to the new matrix in **A**.

2063 It is not an error to call this method more than once on the same variable; however, the handle to
2064 the previously created object will be overwritten.

2065 4.2.5.2 Matrix_dup: Construct a copy of a GraphBLAS matrix

2066 Creates a new matrix with the same domain, dimensions, and contents as another matrix.

2067 C Syntax

```
2068         GrB_Info GrB_Matrix_dup(GrB_Matrix      *C,  
2069                                const GrB_Matrix A);
```

2070 Parameters

2071 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS
2072 matrix.

2073 A (IN) The GraphBLAS matrix to be duplicated.

2074 Return Values

2075 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2076 blocking mode, this indicates that the API checks for the input
2077 arguments passed successfully. Either way, output matrix C is ready
2078 to be used in the next method of the sequence.

2079 GrB_PANIC Unknown internal error.

2080 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2081 GraphBLAS objects (input or output) is in an invalid state caused
2082 by a previous execution error. Call GrB_error() to access any error
2083 messages generated by the implementation.

2084 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2085 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
2086 any matrix constructor.

2087 GrB_NULL_POINTER The C pointer is NULL.

2088 Description

2089 Creates a new matrix **C** of domain **D(A)**, size **nrows(A) × ncols(A)**, and contents **L(A)**. It returns
2090 a handle to it in C.

2091 It is not an error to call this method more than once on the same variable; however, the handle to
2092 the previously created object will be overwritten.

2093 4.2.5.3 Matrix_diag: Construct a diagonal GraphBLAS matrix

2094 Creates a new matrix with the same domain and contents as a GrB_Vector, and square dimensions
2095 appropriate for placing the contents of the vector along the specified diagonal of the matrix.

2096 C Syntax

```
2097         GrB_Info GrB_Matrix_diag(GrB_Matrix      *C,  
2098                                 const GrB_Vector  v,  
2099                                 int64_t          k);
```

2100 Parameters

2101 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS
2102 matrix. The matrix is square with each dimension equal to $\text{size}(\mathbf{v}) + |k|$.

2103 v (IN) The GraphBLAS vector whose contents will be copied to the diagonal of the
2104 matrix.

2105 k (IN) The diagonal to which the vector is assigned. $k = 0$ represents the main
2106 diagonal, $k > 0$ is above the main diagonal, and $k < 0$ is below.

2107 Return Values

2108 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2109 blocking mode, this indicates that the API checks for the input
2110 arguments passed successfully. Either way, output matrix C is ready
2111 to be used in the next method of the sequence.

2112 GrB_PANIC Unknown internal error.

2113 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2114 GraphBLAS objects (input or output) is in an invalid state caused
2115 by a previous execution error. Call GrB_error() to access any error
2116 messages generated by the implementation.

2117 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

2118 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, v, has not been initialized by a call to
2119 Vector_new or Vector_dup.

2120 GrB_NULL_POINTER The C pointer is NULL.

2121 **Description**

2122 Creates a new matrix **C** of domain **D(v)**, size $(\mathbf{size}(\mathbf{v}) + |k|) \times (\mathbf{size}(\mathbf{v}) + |k|)$, and contents

$$2123 \quad \mathbf{L}(\mathbf{C}) = \{(i, i + k, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k \geq 0 \text{ or}$$

$$2124 \quad \mathbf{L}(\mathbf{C}) = \{(i - k, i, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k < 0.$$

2125 It returns a handle to it in **C**. It is not an error to call this method more than once on the same
2126 variable; however, the handle to the previously created object will be overwritten.

2127 **4.2.5.4 Matrix_resize: Resize a matrix**

2128 Changes the dimensions of an existing matrix.

2129 **C Syntax**

```
2130      GrB_Info GrB_Matrix_resize(GrB_Matrix C,  
2131                                GrB_Index  nrows,  
2132                                GrB_Index  ncols);
```

2133 **Parameters**

2134 **C** (INOUT) An existing Matrix object that is being resized.

2135 **nrows** (IN) The new number of rows of the matrix. It can be smaller or larger than the
2136 current number of rows.

2137 **ncols** (IN) The new number of columns of the matrix. It can be smaller or larger than
2138 the current number of columns.

2139 **Return Values**

2140 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
2141 blocking mode, this indicates that the API checks for the input
2142 arguments passed successfully. Either way, output matrix **C** is ready
2143 to be used in the next method of the sequence.

2144 **GrB_PANIC** Unknown internal error.

2145 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2146 GraphBLAS objects (input or output) is in an invalid state caused
2147 by a previous execution error. Call **GrB_error()** to access any error
2148 messages generated by the implementation.

2149 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2150 GrB_NULL_POINTER The C pointer is NULL.

2151 GrB_INVALID_VALUE nrows or ncols is zero or outside the range of the type GrB_Index.

2152 Description

2153 Changes the number of rows and columns of C to nrows and ncols, respectively. The domain $\mathbf{D}(\mathbf{C})$
2154 of matrix C remains the same. The contents $\mathbf{L}(\mathbf{C})$ are modified as described below.

2155 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
2156 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
2157 greater than or equal to ncols are dropped.
2158

2159 4.2.5.5 Matrix_clear: Clear a matrix

2160 Removes all elements (tuples) from a matrix.

2161 C Syntax

2162 GrB_Info GrB_Matrix_clear(GrB_Matrix A);

2163 Parameters

2164 A (IN) An existing GraphBLAS matrix to clear.

2165 Return Values

2166 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2167 blocking mode, this indicates that the API checks for the input ar-
2168 guments passed successfully. Either way, output matrix A is ready
2169 to be used in the next method of the sequence.

2170 GrB_PANIC Unknown internal error.

2171 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2172 GraphBLAS objects (input or output) is in an invalid state caused
2173 by a previous execution error. Call GrB_error() to access any error
2174 messages generated by the implementation.

2175 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2176 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
2177 any matrix constructor.

2178 **Description**

2179 Removes all elements (tuples) from an existing matrix. After the call to `GrB_Matrix_clear(A)`,
2180 $\mathbf{L}(\mathbf{A}) = \emptyset$. The dimensions of the matrix do not change.

2181 **4.2.5.6 Matrix_nrows: Number of rows in a matrix**

2182 Retrieve the number of rows in a matrix.

2183 **C Syntax**

```
2184         GrB_Info GrB_Matrix_nrows(GrB_Index      *nrows,  
2185                                   const GrB_Matrix A);
```

2186 **Parameters**

2187 nrows (OUT) On successful return, contains the number of rows in the matrix.

2188 A (IN) An existing GraphBLAS matrix being queried.

2189 **Return Values**

2190 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
2191 cessfully and the value of `nrows` has been set.

2192 GrB_PANIC Unknown internal error.

2193 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2194 GraphBLAS objects (input or output) is in an invalid state caused
2195 by a previous execution error. Call `GrB_error()` to access any error
2196 messages generated by the implementation.

2197 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to
2198 any matrix constructor.

2199 GrB_NULL_POINTER `nrows` pointer is NULL.

2200 **Description**

2201 Return `nrows(A)` in `nrows` (the number of rows).

2202 **4.2.5.7 Matrix_ncols: Number of columns in a matrix**

2203 Retrieve the number of columns in a matrix.

2204 C Syntax

```
2205         GrB_Info GrB_Matrix_ncols(GrB_Index      *ncols,  
2206                                   const GrB_Matrix A);
```

2207 Parameters

2208 ncols (OUT) On successful return, contains the number of columns in the matrix.

2209 A (IN) An existing GraphBLAS matrix being queried.

2210 Return Values

2211 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
2212 cessfully and the value of ncols has been set.

2213 GrB_PANIC Unknown internal error.

2214 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2215 GraphBLAS objects (input or output) is in an invalid state caused
2216 by a previous execution error. Call GrB_error() to access any error
2217 messages generated by the implementation.

2218 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
2219 any matrix constructor.

2220 GrB_NULL_POINTER ncols pointer is NULL.

2221 Description

2222 Return **ncols(A)** in ncols (the number of columns).

2223 4.2.5.8 Matrix_nvals: Number of stored elements in a matrix

2224 Retrieve the number of stored elements (tuples) in a matrix.

2225 C Syntax

```
2226         GrB_Info GrB_Matrix_nvals(GrB_Index      *nvals,  
2227                                   const GrB_Matrix A);
```

2228 **Parameters**

2229 **nvals** (OUT) On successful return, contains the number of stored elements (tuples) in
2230 the matrix.

2231 **A** (IN) An existing GraphBLAS matrix being queried.

2232 **Return Values**

2233 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2234 cessfully and the value of **nvals** has been set.

2235 **GrB_PANIC** Unknown internal error.

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

2240 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2241 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to
2242 any matrix constructor.

2243 **GrB_NULL_POINTER** The **nvals** pointer is **NULL**.

2244 **Description**

2245 Return **nvals(A)** in **nvals**. This is the number of tuples stored in matrix **A**, which is the size of
2246 **L(A)** (see Section 3.5.3).

2247 **4.2.5.9 Matrix_build: Store elements from tuples into a matrix**

2248 **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);
```

2249 **Parameters**

2250 **C** (INOUT) An existing Matrix object to store the result.

2251 **row_indices** (IN) Pointer to an array of row indices.

2252 **col_indices** (IN) Pointer to an array of column indices.

2253 **values** (IN) Pointer to an array of scalars of a type that is compatible with the domain of
2254 matrix, **C**.

2255 **n** (IN) The number of entries contained in each array (the same for **row_indices**,
2256 **col_indices**, and **values**).

2257 **dup** (IN) An associative and commutative binary operator to apply when duplicate
2258 values for the same location are present in the input arrays. All three domains of
2259 **dup** must be the same; hence $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$. If **dup** is **GrB_NULL**,
2260 then duplicate locations will result in an error.

2261 Return Values

2262 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
2263 blocking mode, this indicates that the API checks for the input
2264 arguments passed successfully. Either way, output matrix **C** is
2265 ready to be used in the next method of the sequence.

2266 **GrB_PANIC** Unknown internal error.

2267 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
2268 opaque GraphBLAS objects (input or output) is in an invalid
2269 state caused by a previous execution error. Call **GrB_error()** to
2270 access any error messages generated by the implementation.

2271 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2272 **GrB_UNINITIALIZED_OBJECT** Either **C** has not been initialized by a call to any matrix construc-
2273 tor, or **dup** has not been initialized by a call to **GrB_BinaryOp_new**.

2274 **GrB_NULL_POINTER** **row_indices**, **col_indices** or **values** pointer is **NULL**.

2275 **GrB_INDEX_OUT_OF_BOUNDS** A value in **row_indices** or **col_indices** is outside the allowed range
2276 for **C**.

2277 **GrB_DOMAIN_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are
2278 not all the same, or the domains of **values** and **C** are incompatible
2279 with each other or D_{dup} .

2280 **GrB_OUTPUT_NOT_EMPTY** Output matrix **C** already contains valid tuples (elements). In
2281 other words, **GrB_Matrix_nvals(C)** returns a positive value.

2282 **GrB_INVALID_VALUE** indices contains a duplicate location and **dup** is **GrB_NULL**.

2283 Description

2284 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
 2285 only differs from \mathbf{C} in its domain; otherwise, $\tilde{\mathbf{C}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \emptyset \rangle$.

2286 Each tuple $\{\text{row_indices}[k], \text{col_indices}[k], \text{values}[k]\}$, where $0 \leq k < n$, is a contribution to the
 2287 output in the form of

$$2288 \quad \tilde{\mathbf{C}}(\text{row_indices}[k], \text{col_indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \text{dup} \neq \text{GrB_NULL} \\ (\mathbf{D}(\mathbf{C})) \text{values}[k] & \text{otherwise.} \end{cases}$$

2289 If multiple values for the same location are present in the input arrays and `dup` is not `GrB_NULL`,
 2290 `dup` is used to reduce the values before assignment into $\tilde{\mathbf{C}}$ as follows:

$$2291 \quad \tilde{\mathbf{C}}_{ij} = \bigoplus_{k: \text{row_indices}[k]=i \wedge \text{col_indices}[k]=j} (D_{dup}) \text{values}[k],$$

2292 where \oplus is the `dup` binary operator. Finally, the resulting $\tilde{\mathbf{C}}$ is copied into \mathbf{C} via typecasting its
 2293 values to $\mathbf{D}(\mathbf{C})$ if necessary. If \oplus is not associative or not commutative, the result is undefined.

2294 The nonopaque input arrays `row_indices`, `col_indices`, and `values` must be at least as large as `n`.

2295 It is an error to call this function on an output object with existing elements. In other words,
 2296 `GrB_Matrix_nvals(C)` should evaluate to zero prior to calling this function.

2297 After `GrB_Matrix_build` returns, it is safe for a programmer to modify or delete the arrays `row_indices`,
 2298 `col_indices`, or `values`.

2299 4.2.5.10 Matrix_setElement: Set a single element in matrix

2300 Set one element of a matrix to a given value.

2301 C Syntax

```
2302 // scalar value
2303 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2304                                <type>         val,
2305                                GrB_Index        row_index,
2306                                GrB_Index        col_index);
2307
2308 // GraphBLAS scalar
2309 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2310                                const GrB_Scalar s,
2311                                GrB_Index        row_index,
2312                                GrB_Index        col_index);
```

2313 Parameters

2314 **C** (INOUT) An existing GraphBLAS matrix for which an element is to be assigned.
2315 **val** or **s** (IN) Scalar to assign. Its domain (type) must be compatible with the domain of
2316 **C**.
2317 **row_index** (IN) Row index of element to be assigned
2318 **col_index** (IN) Column index of element to be assigned

2319 Return Values

2320 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
2321 blocking mode, this indicates that the compatibility tests on in-
2322 dex/dimensions and domains for the input arguments passed suc-
2323 cessfully. Either way, the output matrix **C** is ready to be used in
2324 the next method of the sequence.
2325 **GrB_PANIC** Unknown internal error.
2326 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2327 GraphBLAS objects (input or output) is in an invalid state caused
2328 by a previous execution error. Call **GrB_error()** to access any error
2329 messages generated by the implementation.
2330 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.
2331 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, or GraphBLAS scalar, **s**, has not been
2332 initialized by a call to a respective constructor.
2333 **GrB_INVALID_INDEX** **row_index** or **col_index** is outside the allowable range (i.e., not less
2334 than **nrows(C)** or **ncols(C)**, respectively).
2335 **GrB_DOMAIN_MISMATCH** The domains of the matrix and the scalar are incompatible.

2336 Description

2337 First, the scalar and output matrix are tested for domain compatibility as follows: **D(val)** or
2338 **D(s)** must be compatible with **D(C)**. Two domains are compatible with each other if values from
2339 one domain can be cast to values in the other domain as per the rules of the C language. In
2340 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
2341 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
2342 **GrB_Matrix_setElement** ends and the domain mismatch error listed above is returned.

2343 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2344 \quad & 0 \leq \text{row_index} < \text{nrows}(\mathbf{C}), \\ & 0 \leq \text{col_index} < \text{ncols}(\mathbf{C}) \end{aligned}$$

2345 If either of these conditions is violated, execution of `GrB_Matrix_setElement` ends and the invalid
 2346 index error listed above is returned.

2347 We are now ready to carry out the assignment; that is:

$$2348 \quad C(\text{row_index}, \text{col_index}) = \begin{cases} \mathbf{L}(\mathbf{s}), & \text{GraphBLAS scalar.} \\ \text{val}, & \text{otherwise.} \end{cases}$$

2349 In the case of a transparent scalar or if $\mathbf{L}(\mathbf{s})$ is not empty, then a value will be stored at the
 2350 specified location in \mathbf{C} , overwriting any value that may have been stored there before. In the case
 2351 of a GraphBLAS scalar and if $\mathbf{L}(\mathbf{s})$ is empty, then any value stored at the specified location in \mathbf{C}
 2352 will be removed.

2353 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents
 2354 of \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with
 2355 return value `GrB_SUCCESS` and the new content of vector \mathbf{C} is as defined above but may not be
 2356 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

2357 **4.2.5.11 `Matrix_removeElement`: Remove an element from a matrix**

2358 Remove (annihilate) one stored element from a matrix.

2359 **C Syntax**

```
2360      GrB_Info GrB_Matrix_removeElement(GrB_Matrix  C,
2361                                       GrB_Index   row_index,
2362                                       GrB_Index   col_index);
```

2363 **Parameters**

2364 `C` (INOUT) An existing GraphBLAS matrix from which an element is to be removed.

2365 `row_index` (IN) Row index of element to be removed

2366 `col_index` (IN) Column index of element to be removed

2367 **Return Values**

2368 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
 2369 blocking mode, this indicates that the compatibility tests on in-
 2370 dex/dimensions and domains for the input arguments passed suc-
 2371 cessfully. Either way, the output matrix \mathbf{C} is ready to be used in
 2372 the next method of the sequence.

2373 `GrB_PANIC` Unknown internal error.

2374 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 2375 GraphBLAS objects (input or output) is in an invalid state caused
 2376 by a previous execution error. Call GrB_error() to access any error
 2377 messages generated by the implementation.

2378 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2379 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, C, has not been initialized by a call to
 2380 any matrix constructor.

2381 GrB_INVALID_INDEX row_index or col_index is outside the allowable range (i.e., not less
 2382 than **nrows(C)** or **ncols(C)**, respectively).

2383 Description

2384 First, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned}
 &0 \leq \text{row_index} < \text{nrows}(\mathbf{C}), \\
 &0 \leq \text{col_index} < \text{ncols}(\mathbf{C})
 \end{aligned}$$

2386 If either of these conditions is violated, execution of GrB_Matrix_removeElement ends and the
 2387 invalid index error listed above is returned.

2388 We are now ready to carry out the removal of a value that may be stored at the location specified by
 2389 (row_index, col_index). If a value does not exist at the specified location in C, no error is reported
 2390 and the operation has no effect on the state of C. In either case, the following will be true on return
 2391 from this method: (row_index, col_index) \notin ind(C)

2392 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new contents
 2393 of C is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with
 2394 return value GrB_SUCCESS and the new content of vector C is as defined above but may not be
 2395 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

2396 4.2.5.12 Matrix_extractElement: Extract a single element from a matrix

2397 Extract one element of a matrix into a scalar.

2398 C Syntax

```

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

```

2406         GrB_Info GrB_Matrix_extractElement(GrB_Scalar      s,
2407                                             const GrB_Matrix A,
2408                                             GrB_Index      row_index,
2409                                             GrB_Index      col_index);
2410

```

2411 Parameters

2412 **val or s (INOUT)** An existing scalar whose domain is compatible with the domain of matrix
2413 **A.** On successful return, this scalar holds the result of the extract. Any previous
2414 value stored in **val** or **s** is overwritten.

2415 **A (IN)** The GraphBLAS matrix from which an element is extracted.

2416 **row_index (IN)** The row index of location in **A** to extract.

2417 **col_index (IN)** The column index of location in **A** to extract.

2418 Return Values

2419 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2420 cessfully. This indicates that the compatibility tests on dimensions
2421 and domains for the input arguments passed successfully, and the
2422 output scalar, **val** or **s**, has been computed and is ready to be used
2423 in the next method of the sequence.

2424 **GrB_NO_VALUE** When using the transparent scalar, **val**, this is returned when there
2425 is no stored value at specified location.

2426 **GrB_PANIC** Unknown internal error.

2427 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2428 GraphBLAS objects (input or output) is in an invalid state caused
2429 by a previous execution error. Call **GrB_error()** to access any error
2430 messages generated by the implementation.

2431 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2432 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, or scalar, **s**, has not been initialized by
2433 a call to a corresponding constructor.

2434 **GrB_NULL_POINTER** **val** pointer is **NULL**.

2435 **GrB_INVALID_INDEX** **row_index** or **col_index** is outside the allowable range (i.e. less than
2436 zero or greater than or equal to **nrows(A)** or **ncols(A)**, respec-
2437 tively).

2438 **GrB_DOMAIN_MISMATCH** The domains of the matrix and scalar are incompatible.

2439 Description

2440 First, the scalar and input matrix are tested for domain compatibility as follows: $\mathbf{D}(\text{val})$ or $\mathbf{D}(\mathbf{s})$
 2441 must be compatible with $\mathbf{D}(\mathbf{A})$. Two domains are compatible with each other if values from
 2442 one domain can be cast to values in the other domain as per the rules of the C language. In
 2443 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
 2444 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
 2445 `GrB_Matrix_extractElement` ends and the domain mismatch error listed above is returned.

2446 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2447 \quad & 0 \leq \text{row_index} < \mathbf{nrows}(\mathbf{A}), \\ & 0 \leq \text{col_index} < \mathbf{ncols}(\mathbf{A}) \end{aligned}$$

2448 If either condition is violated, execution of `GrB_Matrix_extractElement` ends and the invalid index
 2449 error listed above is returned.

2450 We are now ready to carry out the extract into the output scalar; that is,

$$2451 \quad \left. \begin{array}{l} \mathbf{L}(\mathbf{s}) \\ \text{val} \end{array} \right\} = \mathbf{A}(\text{row_index}, \text{col_index})$$

2452 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 val
 2453 with casting as necessary. If $(\text{row_index}, \text{col_index}) \notin \mathbf{ind}(\mathbf{A})$, then one of the follow occurs
 2454 depending on output scalar type:

- 2455 • The GraphBLAS scalar, \mathbf{s} , is cleared and `GrB_SUCCESS` is returned.
- 2456 • The non-opaque scalar, val , is unchanged, and `GrB_NO_VALUE` is returned.

2457 When using the non-opaque scalar variant (val) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`
 2458 mode, the new contents of val are as defined above if the method exits with return value `GrB_SUCCESS`
 2459 or `GrB_NO_VALUE`.

2460 When using the GraphBLAS scalar variant (\mathbf{s}) with a `GrB_SUCCESS` return value, the method
 2461 exits and the new contents of \mathbf{s} is as defined above and fully computed in `GrB_BLOCKING` mode.
 2462 In `GrB_NONBLOCKING` mode, the new contents of \mathbf{s} is as defined above but may not be fully
 2463 computed; however, it can be used in the next GraphBLAS method call in a sequence.

2464 4.2.5.13 Matrix_extractTuples: Extract tuples from a matrix

2465 Extract the contents of a GraphBLAS matrix into non-opaque data structures.

2466 C Syntax

```
2467      GrB_Info GrB_Matrix_extractTuples(GrB_Index      *row_indices,
2468                                      GrB_Index      *col_indices,
```

```

2469                                     <type>                *values,
2470                                     GrB_Index                *n,
2471                                     const GrB_Matrix          A);

```

2472 Parameters

2473 **row_indices** (OUT) Pointer to an array of row indices that is large enough to hold all of the
2474 row indices.

2475 **col_indices** (OUT) Pointer to an array of column indices that is large enough to hold all of the
2476 column indices.

2477 **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of
2478 the stored values whose type is compatible with $\mathbf{D}(\mathbf{A})$.

2479 **n** (INOUT) Pointer to a value indicating (in input) the number of elements the **values**,
2480 **row_indices**, and **col_indices** arrays can hold. Upon return, it will contain the
2481 number of values written to the arrays.

2482 **A** (IN) An existing GraphBLAS matrix.

2483 Return Values

2484 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2485 cessfully. This indicates that the compatibility tests on the input
2486 argument passed successfully, and the output arrays, **indices** and
2487 **values**, have been computed.

2488 **GrB_PANIC** Unknown internal error.

2489 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2490 GraphBLAS objects (input or output) is in an invalid state caused
2491 by a previous execution error. Call **GrB_error()** to access any error
2492 messages generated by the implementation.

2493 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2494 **GrB_INSUFFICIENT_SPACE** Not enough space in **row_indices**, **col_indices**, and **values** (as indi-
2495 cated by the **n** parameter) to hold all of the tuples that will be
2496 extracted.

2497 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to
2498 any matrix constructor.

2499 **GrB_NULL_POINTER** **row_indices**, **col_indices**, **values** or **n** pointer is NULL.

2500 **GrB_DOMAIN_MISMATCH** The domains of the **A** matrix and **values** array are incompatible
2501 with one another.

2502 Description

2503 This method will extract all the tuples from the GraphBLAS matrix **A**. The values associated with
2504 those tuples are placed in the **values** array, the column indices are placed in the **col_indices** array,
2505 and the row indices are placed in the **row_indices** array. These output arrays are pre-allocated by
2506 the user before calling this function such that each output array has enough space to hold at least
2507 **GrB_Matrix_nvals(A)** elements.

2508 Upon return of this function, a pair of $\{\text{row_indices}[k], \text{col_indices}[k]\}$ are unique for every valid
2509 k , but they are not required to be sorted in any particular order. Each tuple (i, j, A_{ij}) in **A** is
2510 unzipped and copied into a distinct k th location in output vectors:

$$\{\text{row_indices}[k], \text{col_indices}[k], \text{values}[k]\} \leftarrow (i, j, A_{ij}),$$

2511 where $0 \leq k < \text{GrB_Matrix_nvals}(v)$. No gaps in output vectors are allowed; that is, if **row_indices**[k],
2512 **col_indices**[k] and **values**[k] exist upon return, so does **row_indices**[j], **col_indices**[j] and **values**[j] for
2513 all j such that $0 \leq j < k$.

2514 Note that if the value in **n** on input is less than the number of values contained in the matrix **A**,
2515 then a **GrB_INSUFFICIENT_SPACE** error is returned since it is undefined which subset of values
2516 would be extracted.

2517 In both **GrB_BLOCKING** mode **GrB_NONBLOCKING** mode if the method exits with return value
2518 **GrB_SUCCESS**, the new contents of the arrays **row_indices**, **col_indices** and **values** are as defined
2519 above.

2520 4.2.5.14 Matrix_exportHint: Provide a hint as to which storage format might be most 2521 efficient for exporting a matrix

2522 C Syntax

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

2523 Parameters

2524 **hint** (OUT) Pointer to a value of type **GrB_Format**.

2525 **A** (IN) A GraphBLAS matrix object.

2526 Return Values

2527 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2528 cessfully and the value of **hint** has been set.

2529 **GrB_PANIC** Unknown internal error.

2530 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
2531 opaque GraphBLAS objects (input or output) is in an invalid
2532 state caused by a previous execution error. Call GrB_error() to
2533 access any error messages generated by the implementation.

2534 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2535 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
2536 any matrix constructor.

2537 GrB_NULL_POINTER hint is NULL.

2538 GrB_NO_VALUE If the implementation does not have a preferred format, it may
2539 return the value GrB_NO_VALUE.

2540 Description

2541 Given a GraphBLAS matrix A, provide a hint as to which format might be most efficient for
2542 exporting the matrix A. GraphBLAS implementations might return the current storage format of
2543 the matrix, or the format to which it could most efficiently be exported. However, implementations
2544 are free to return any value for format defined in Section 3.5.3.1. Note that an implementation is
2545 free to refuse to provide a format hint, returning GrB_NO_VALUE.

2546 4.2.5.15 Matrix_exportSize: Return the array sizes necessary to export a GraphBLAS 2547 matrix object

2548 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);

```

2549 Parameters

2550 n_indptr (OUT) Pointer to a value of type GrB_Index.

2551 n_indices (OUT) Pointer to a value of type GrB_Index.

2552 n_values (OUT) Pointer to a value of type GrB_Index.

2553 format (IN) a value indicating the format in which the matrix will be exported, as defined
2554 in Section 3.5.3.1.

2555 A (IN) A GraphBLAS matrix object.

2556 Return Values

2557 GrB_SUCCESS In blocking mode or non-blocking mode, the operation com-
2558 pleted successfully. This indicates that the API checks for the
2559 input arguments passed successfully, and the number of elements
2560 necessary for the export buffers have been written to `n_indptr`,
2561 `n_indices`, and `n_values`, respectively.

2562 GrB_PANIC Unknown internal error.

2563 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
2564 opaque GraphBLAS objects (input or output) is in an invalid
2565 state caused by a previous execution error. Call `GrB_error()` to
2566 access any error messages generated by the implementation.

2567 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2568 GrB_UNINITIALIZED_OBJECT The GraphBLAS Matrix, `A`, has not been initialized by a call to
2569 any matrix constructor.

2570 GrB_NULL_POINTER `n_indptr`, `n_indices`, or `n_values` is NULL.

2571 Description

2572 Given a matrix `A`, returns the required capacities of arrays `values`, `indptr`, and `indices` necessary to
2573 export the matrix in the format specified by `format`. The output values `n_values`, `n_indptr`, and
2574 `indices` will contain the corresponding sizes of the arrays (in number of elements) that must be
2575 allocated to hold the exported matrix. The argument `format` can be chosen arbitrarily by the user
2576 as one of the values defined in Section 3.5.3.1.

2577 4.2.5.16 Matrix_export: Export a GraphBLAS matrix to a pre-defined format

2578 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);
```

2579 Parameters

2580 **indptr** (INOUT) Pointer to an array that will hold row or column offsets, or row in-
2581 dices, depending on the value of **format**. It must be large enough to hold at
2582 least **n_indptr** elements of type **GrB_Index**, where **n_indices** was returned from
2583 **GrB_Matrix_exportSize()** method.

2584 **indices** (INOUT) Pointer to an array that will hold row or column indices of the elements
2585 in **values**, depending on the value of **format**. It must be large enough to hold at
2586 least **n_indices** elements of type **GrB_Index**, where **n_indices** was returned from
2587 **GrB_Matrix_exportSize()** method.

2588 **values** (INOUT) Pointer to an array that will hold stored values. The type of ele-
2589 ment must match the type of the values stored in **A**. It must be large enough
2590 to hold at least **n_values** elements of that type, where **n_values** was returned from
2591 **GrB_Matrix_exportSize**.

2592 **n_indptr** (INOUT) Pointer to a value indicating (on input) the number of elements the **indptr**
2593 array can hold. Upon return, it will contain the number of elements written to the
2594 array.

2595 **n_indices** (INOUT) Pointer to a value indicating (on input) the number of elements the **indices**
2596 array can hold. Upon return, it will contain the number of elements written to the
2597 array.

2598 **n_values** (INOUT) Pointer to a value indicating (on input) the number of elements the **values**
2599 array can hold. Upon return, it will contain the number of elements written to the
2600 array.

2601 **format** (IN) a value indicating the format in which the matrix will be exported, as defined
2602 in Section 3.5.3.1.

2603 **A** (IN) A GraphBLAS matrix object.

2604 Return Values

2605 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2606 cessfully. This indicates that the compatibility tests on the input
2607 argument passed successfully, and the output arrays, **indptr**, **in-**
2608 **dices** and **values**, have been computed.

2609 **GrB_PANIC** Unknown internal error.

2610 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
2611 opaque GraphBLAS objects (input or output) is in an invalid
2612 state caused by a previous execution error. Call **GrB_error()** to
2613 access any error messages generated by the implementation.

2614 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2639 **nrows** (IN) Integer value holding the number of rows in the matrix.

2640 **ncols** (IN) Integer value holding the number of columns in the matrix.

2641 **indptr** (IN) Pointer to an array of row or column offsets, or row indices, depending on the
2642 value of **format**.

2643 **indices** (IN) Pointer to an array row or column indices of the elements in **values**, depending
2644 on the value of **format**.

2645 **values** (IN) Pointer to an array of values. Type must match the type of **d**.

2646 **n_indptr** (IN) Integer value holding the number of elements in the array pointed to by **indptr**.

2647 **n_indices** (IN) Integer value holding the number of elements in the array pointed to by **indices**.

2648 **n_values** (IN) Integer value holding the number of elements in the array pointed to by **values**.

2649 **format** (IN) a value indicating the format of the matrix being imported, as defined in
2650 Section 3.5.3.1.

2651 **Return Values**

2652 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
2653 blocking mode, this indicates that the API checks for the input
2654 arguments passed successfully and the input arrays have been
2655 consumed. Either way, output matrix **A** is ready to be used in
2656 the next method of the sequence.

2657 **GrB_PANIC** Unknown internal error.

2658 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2659 **GrB_UNINITIALIZED_OBJECT** The **GrB_Type** object has not been initialized by a call to **GrB_Type_new**
2660 (needed for user-defined types).

2661 **GrB_NULL_POINTER** **A**, **indptr**, **indices** or **values** pointer is **NULL**.

2662 **GrB_INDEX_OUT_OF_BOUNDS** A value in **indptr** or **indices** is outside the allowed range for indices
2663 in **A** and or the size of **values**, **n_values**, depending on the value
2664 of **format**.

2665 **GrB_INVALID_VALUE** **nrows** or **ncols** is zero or outside the range of the type **GrB_Index**.

2666 **GrB_DOMAIN_MISMATCH** The domain given in parameter **d** does not match the element
2667 type of **values**.

2668 Description

2669 Creates a new matrix **A** of domain **D**(d) and dimension **nrows** \times **ncols**. The new GraphBLAS
2670 matrix will be filled with the contents of the matrix pointed to by **indptr**, and **indices**, and **values**.
2671 The method returns a handle to the new matrix in **A**. The structure of the data being imported is
2672 defined by **format**, which must be equal to one of the values defined in Section 3.5.3.1. Details of
2673 the contents of **indptr**, **indices** and **values** for each supported format is given in Appendix B.

2674 It is not an error to call this method more than once on the same output matrix; however, the
2675 handle to the previously created object will be overwritten.

2676 4.2.5.18 Matrix_serializeSize: Compute the serialize buffer size

2677 Compute the buffer size (in bytes) necessary to serialize a GrB_Matrix using GrB_Matrix_serialize.

2678 C Syntax

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

2679 Parameters

2680 size (OUT) Pointer to GrB_Index value where size in bytes of serialized object will be
2681 written.

2682 A (IN) A GraphBLAS matrix object.

2683 Return Values

2684 GrB_SUCCESS The operation completed successfully and the value pointed to
2685 by *size has been computed and is ready to use.

2686 GrB_PANIC Unknown internal error.

2687 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2688 GrB_NULL_POINTER size is NULL.

2689 Description

2690 Returns the size in bytes of the data buffer necessary to serialize the GraphBLAS matrix object A.
2691 Users may then allocate a buffer of size bytes to pass as a parameter to GrB_Matrix_serialize.

2692 **4.2.5.19 Matrix_serialize: Serialize a GraphBLAS matrix.**

2693 Serialize a GraphBLAS Matrix object into an opaque stream of bytes.

2694 **C Syntax**

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

2695 **Parameters**

2696 **serialized_data** (INOUT) Pointer to the preallocated buffer where the serialized matrix will be
2697 written.

2698 **serialized_size** (INOUT) On input, the size in bytes of the buffer pointed to by **serialized_data**.
2699 On output, the number of bytes written to **serialized_data**.

2700 **A** (IN) A GraphBLAS matrix object.

2701 **Return Values**

2702 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2703 cessfully. This indicates that the compatibility tests on the in-
2704 put argument passed successfully, and the output buffer **serial-
2705 ized_data** and **serialized_size**, have been computed and are ready
2706 to use.

2707 **GrB_PANIC** Unknown internal error.

2708 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
2709 opaque GraphBLAS objects (input or output) is in an invalid
2710 state caused by a previous execution error. Call **GrB_error()** to
2711 access any error messages generated by the implementation.

2712 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2713 **GrB_NULL_POINTER** **serialized_data** or **serialize_size** is NULL.

2714 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to
2715 any matrix constructor.

2716 **GrB_INSUFFICIENT_SPACE** The size of the buffer **serialized_data** (provided as an input **seri-
2717 alized_size**) was not large enough.

2718 Description

2719 Serializes a GraphBLAS matrix object to an opaque buffer. To guarantee successful execution,
2720 the size of the buffer pointed to by `serialized_data`, provided as an input by `serialized_size`, must
2721 be of at least the number of bytes returned from `GrB_Matrix_serializeSize`. The actual size of the
2722 serialized matrix written to `serialized_data` is provided upon completion as an output written to
2723 `serialized_size`.

2724 The contents of the serialized buffer are implementation defined. Thus, a serialized matrix created
2725 with one library implementation is not necessarily valid for deserialization with another implemen-
2726 tation.

2727 4.2.5.20 Matrix_deserialize: Deserialize a GraphBLAS matrix.

2728 Construct a new GraphBLAS matrix from a serialized object.

2729 C Syntax

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

2730 Parameters

2731 A (INOUT) On a successful return, contains a handle to the newly created Graph-
2732 BLAS matrix.

2733 d (IN) the type of the matrix that was serialized in `serialized_data`.
2734 If d is `GrB_NULL`, the implementation must attempt to deserialize the matrix
2735 without a provided type object.

2736 `serialized_data` (IN) a pointer to a serialized GraphBLAS matrix created with `GrB_Matrix_serialize`.

2737 `serialized_size` (IN) the size of the buffer pointed to by `serialized_data` in bytes.

2738 Return Values

2739 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
2740 blocking mode, this indicates that the API checks for the input
2741 arguments passed successfully. Either way, output matrix A is
2742 ready to be used in the next method of the sequence.

2743 `GrB_PANIC` Unknown internal error.

2744 `GrB_INVALID_OBJECT` This is returned if `serialized_data` is invalid or corrupted.

2772 **Return Value**

2773 GrB_SUCCESS The method completed successfully.

2774 GrB_PANIC unknown internal error.

2775 GrB_OUT_OF_MEMORY not enough memory available for operation.

2776 GrB_NULL_POINTER desc pointer is NULL.

2777 **Description**

2778 Creates a new descriptor object and returns a handle to it in **desc**. A newly created descriptor can
2779 be populated by calls to **Descriptor_set**.

2780 It is not an error to call this method more than once on the same variable; however, the handle to
2781 the previously created object will be overwritten.

2782 **4.2.6.2 Descriptor_set: Set content of descriptor**

2783 Sets the content for a field for an existing descriptor.

2784 **C Syntax**

```
2785           GrB_Info GrB_Descriptor_set(GrB_Descriptor        desc,  
2786                                       GrB_Desc_Field        field,  
2787                                       GrB_Desc_Value        val);
```

2788 **Parameters**

2789 **desc** (IN) An existing GraphBLAS descriptor to be modified.

2790 **field** (IN) The field being set.

2791 **val** (IN) New value for the field being set.

2792 **Return Values**

2793 GrB_SUCCESS operation completed successfully.

2794 GrB_PANIC unknown internal error.

2795 GrB_OUT_OF_MEMORY not enough memory available for operation.

2796 GrB_UNINITIALIZED_OBJECT the desc parameter has not been initialized by a call to **new**.

2797 GrB_INVALID_VALUE invalid value set on the field, or invalid field.

2798 Description

2799 For a given descriptor, the `GrB_Descriptor_set` method can be called for each field in the descriptor
2800 to set the value associated with that field. Valid values for the `field` parameter include the following:

2801 `GrB_OUTP` refers to the output parameter (result) of the operation.

2802 `GrB_MASK` refers to the mask parameter of the operation.

2803 `GrB_INP0` refers to the first input parameters of the operation (matrices and vectors).

2804 `GrB_INP1` refers to the second input parameters of the operation (matrices and vectors).

2805 Valid values for the `val` parameter are:

2806 `GrB_STRUCTURE` Use only the structure of the stored values of the corresponding mask
2807 (`GrB_MASK`) parameter.

2808 `GrB_COMP` Use the complement of the corresponding mask (`GrB_MASK`) param-
2809 eter. When combined with `GrB_STRUCTURE`, the complement of the
2810 structure of the mask is used without evaluating the values stored.

2811 `GrB_TRAN` Use the transpose of the corresponding matrix parameter (valid for input
2812 matrix parameters only).

2813 `GrB_REPLACE` When assigning the masked values to the output matrix or vector, clear
2814 the matrix first (or clear the non-masked entries). The default behavior
2815 is to leave non-masked locations unchanged. Valid for the `GrB_OUTP`
2816 parameter only.

2817 Descriptor values can only be set, and once set, cannot be cleared. As, in the case of `GrB_MASK`,
2818 multiple values can be set and all will apply (for example, both `GrB_COMP` and `GrB_STRUCTURE`).
2819 A value for a given field may be set multiple times but will have no additional effect. Fields that
2820 have no values set result in their default behavior, as defined in Section 3.6.

2821 4.2.7 free: Destroy an object and release its resources

2822 Destroys a previously created GraphBLAS object and releases any resources associated with the
2823 object.

2824 C Syntax

2825 `GrB_Info GrB_free(<GrB_Object> *obj);`

2826 Parameters

2827 **obj** (INOUT) An existing GraphBLAS object to be destroyed. The object must have
2828 been created by an explicit call to a GraphBLAS constructor. It can be any of the
2829 opaque GraphBLAS objects such as matrix, vector, descriptor, semiring, monoid,
2830 binary op, unary op, or type. On successful completion of **GrB_free**, **obj** behaves
2831 as an uninitialized object.

2832 Return Values

2833 **GrB_SUCCESS** operation completed successfully

2834 **GrB_PANIC** unknown internal error. If this return value is encountered when
2835 in nonblocking mode, the error responsible for the panic condition
2836 could be from any method involved in the computation of the input
2837 object. The **GrB_error()** method should be called for additional
2838 information.

2839 Description

2840 GraphBLAS objects consume memory and other resources managed by the GraphBLAS runtime
2841 system. A call to **GrB_free** frees those resources so they are available for use by other GraphBLAS
2842 objects.

2843 The parameter passed into **GrB_free** is a handle referencing a GraphBLAS opaque object of a data
2844 type from table 2.1. The object must have been created by an explicit call to a GraphBLAS con-
2845 structor. The behavior of a program that calls **GrB_free** on a pre-defined object is implementation
2846 defined.

2847 After the **GrB_free** method returns, the object referenced by the input handle is destroyed and the
2848 handle has the value **GrB_INVALID_HANDLE**. The handle can be used in subsequent GraphBLAS
2849 methods but only after the handle has been reinitialized with a call the the appropriate **_new** or
2850 **_dup** method.

2851 Note that unlike other GraphBLAS methods, calling **GrB_free** with an object with an invalid handle
2852 is legal. The system may attempt to free resources that might be associated with that object, if
2853 possible, and return normally.

2854 When using **GrB_free** it is possible to create a dangling reference to an object. This would occur
2855 when a handle is assigned to a second variable of the same opaque type. This creates two handles
2856 that reference the same object. If **GrB_free** is called with one of the variables, the object is destroyed
2857 and the handle associated with the other variable no longer references a valid object. This is not an
2858 error condition that the implementation of the GraphBLAS API can be expected to catch, hence
2859 programmers must take care to prevent this situation from occurring.

2860 **4.2.8 wait: Return once an object is either *complete* or *materialized***

2861 Wait until method calls in a sequence put an object into a state of *completion* or *materialization*.

2862 **C Syntax**

2863 `GrB_Info GrB_wait(GrB_Object obj, GrB_WaitMode mode);`

2864 **Parameters**

2865 `obj` (INOUT) An existing GraphBLAS object. The object must have been created by an
2866 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS
2867 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,
2868 or type. On successful return of `GrB_wait`, the `obj` can be safely read from another
2869 thread (completion) or all computing to produce `obj` by all GraphBLAS operations
2870 in its sequence have finished (materialization).

2871 `mode` (IN) Set's the mode for `GrB_wait` for whether it is waiting for `obj` to be in the
2872 state of *completion* or *materialization*. Acceptable values are `GrB_COMPLETE` or
2873 `GrB_MATERIALIZE`.

2874 **Return values**

2875 `GrB_SUCCESS` operation completed successfully.

2876 `GrB_INDEX_OUT_OF_BOUNDS` an index out-of-bounds execution error happened during com-
2877 pletion of pending operations.

2878 `GrB_OUT_OF_MEMORY` and out-of-memory execution error happened during completion
2879 of pending operations.

2880 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`,
2881 or other constructor, method.

2882 `GrB_PANIC` unknown internal error.

2883 `GrB_INVALID_VALUE` method called with a `GrB_WaitMode` other than `GrB_COMPLETE`
2884 `GrB_MATERIALIZE`.

2885 **Description**

2886 On successful return from `GrB_wait()`, the input object, `obj` is in one of two states depending on
2887 the mode of `GrB_wait`:

- 2888 • *complete*: `obj` can be used in a happens-before relation, so in a properly synchronized program
2889 it can be safely used as an IN or INOUT parameter in a GraphBLAS method call from another
2890 thread. This result occurs when the mode parameter is set to `GrB_COMPLETE`.
- 2891 • *materialized*: `obj` is *complete*, but in addition, no further computing will be carried out on
2892 behalf of `obj` and error information is available. This result occurs when the mode parameter
2893 is set to `GrB_MATERIALIZE`.

2894 Since in blocking mode OUT or INOUT parameters to any method call are materialized upon return,
2895 `GrB_wait(obj,mode)` has no effect when called in blocking mode.

2896 In non-blocking mode, the status of any pending method calls, other than those associated with pro-
2897 ducing the *complete* or *materialized* state of `obj`, are not impacted by the call to `GrB_wait(obj,mode)`.
2898 Methods in the sequence for `obj`, however, most likely would be impacted by a call to `GrB_wait(obj,mode)`;
2899 especially in the case of the *materialized* mode for which any computing on behalf of `obj` must be
2900 finished prior to the return from `GrB_wait(obj,mode)`.

2901 4.2.9 error: Retrieve an error string

2902 Retrieve an error-message about any errors encountered during the processing associated with an
2903 object.

2904 C Syntax

```
2905         GrB_Info GrB_error(const char          **error,
2906                           const GrB_Object      obj);
```

2907 Parameters

2908 `error` (OUT) A pointer to a null-terminated string. The contents of the string are im-
2909 plementation defined.

2910 `obj` (IN) An existing GraphBLAS object. The object must have been created by an
2911 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS
2912 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,
2913 or type.

2914 Return value

2915 `GrB_SUCCESS` operation completed successfully.

2916 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`,
2917 or other constructor, method.

2918 `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	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A} \otimes \mathbf{B}$

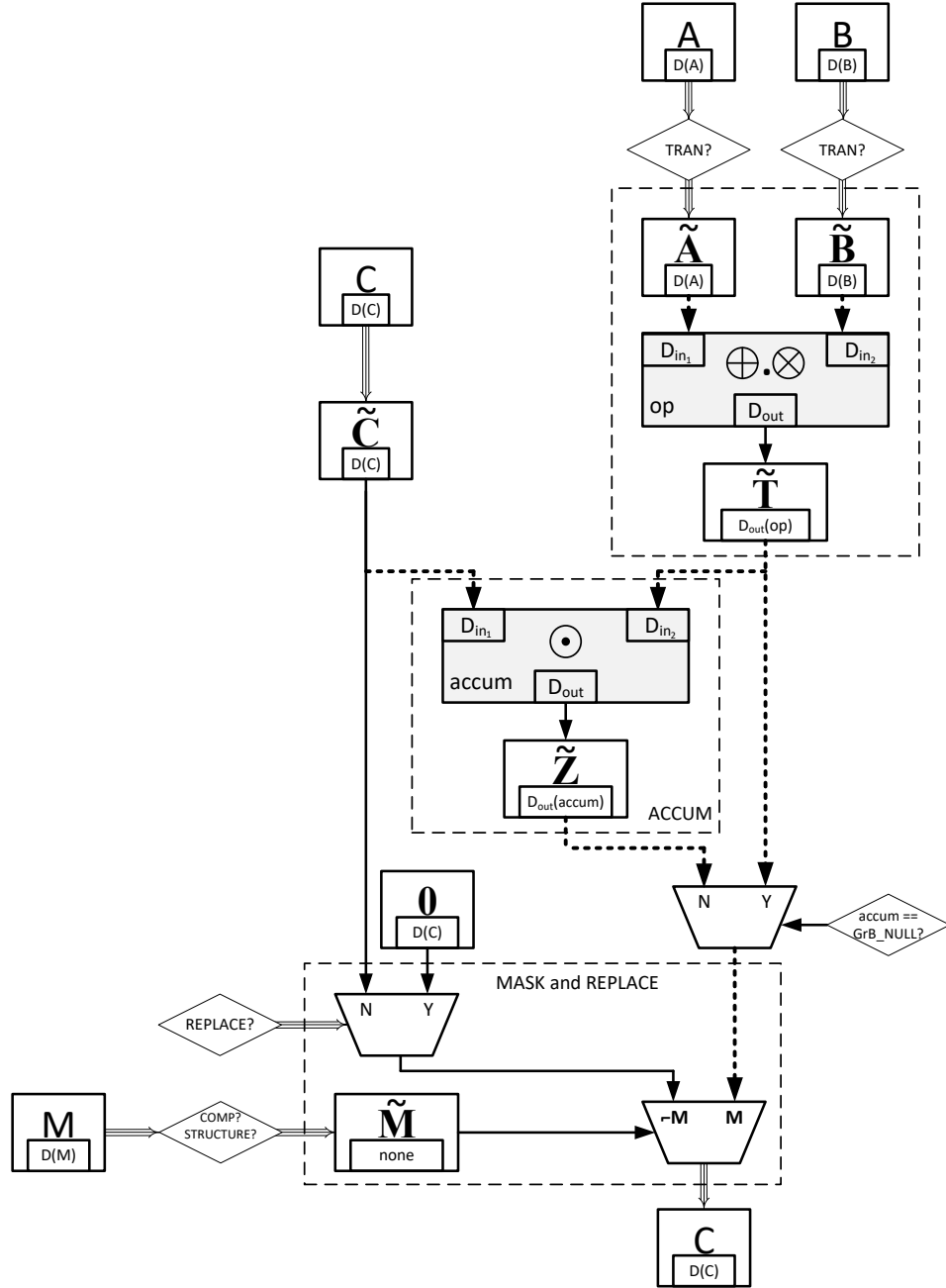


Figure 4.1: Flowchart for the GraphBLAS operations. Although shown specifically for the mxm operation, many elements are common to all operations: such as the “ACCUM” and “MASK and REPLACE” blocks. The triple arrows (\Rightarrow) denote where “as if copy” takes place (including both collections and descriptor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

2988 To support the detection of as many cases of uninitialized objects as possible, it is strongly rec-
 2989 ommended to initialize all GraphBLAS objects to the predefined value `GrB_INVALID_HANDLE` at
 2990 the point of their declaration, as shown in the following examples:

```
2991      GrB_Type      type = GrB_INVALID_HANDLE;
2992      GrB_Semiring  semiring = GrB_INVALID_HANDLE;
2993      GrB_Matrix    matrix = GrB_INVALID_HANDLE;
```

2994 Compliance

2995 We follow a *prescriptive* approach to the definition of the semantics of GraphBLAS operations.
 2996 That is, for each operation we give a recipe for producing its outcome. Any implementation that
 2997 produces the same outcome, and follows the GraphBLAS execution model (Section 2.5) and error
 2998 model (Section 2.6) is a conforming implementation.

2999 4.3.1 mxm: Matrix-matrix multiply

3000 Multiplies a matrix with another matrix on a semiring. The result is a matrix.

3001 C Syntax

```
3002      GrB_Info GrB_mxm(GrB_Matrix      C,
3003                      const GrB_Matrix Mask,
3004                      const GrB_BinaryOp accum,
3005                      const GrB_Semiring op,
3006                      const GrB_Matrix A,
3007                      const GrB_Matrix B,
3008                      const GrB_Descriptor desc);
```

3009 Parameters

3010 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
 3011 that may be accumulated with the result of the matrix product. On output, the
 3012 matrix holds the results of the operation.

3013 **Mask** (IN) An optional “write” mask that controls which results from this operation are
 3014 stored into the output matrix C. The mask dimensions must match those of the
 3015 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
 3016 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types
 3017 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
 3018 dimensions of C), `GrB_NULL` should be specified.

3019 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
 3020 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
 3021 specified.

3022 **op** (IN) The semiring used in the matrix-matrix multiply.

3023 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 3024 multiplication.

3025 **B** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
 3026 multiplication.

3027 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
 3028 should be specified. Non-default field/value pairs are listed as follows:
 3029

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.

3031 Return Values

3032 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 3033 blocking mode, this indicates that the compatibility tests on di-
 3034 mensions and domains for the input arguments passed successfully.
 3035 Either way, output matrix **C** is ready to be used in the next method
 3036 of the sequence.

3037 **GrB_PANIC** Unknown internal error.

3038 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 3039 GraphBLAS objects (input or output) is in an invalid state caused
 3040 by a previous execution error. Call **GrB_error()** to access any error
 3041 messages generated by the implementation.

3042 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

3043 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 3044 a call to **new** (or **Matrix_dup** for matrix parameters).

3045 **GrB_DIMENSION_MISMATCH** Mask and/or matrix dimensions are incompatible.

3046 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 3047 corresponding domains of the semiring or accumulation operator,
 3048 or the mask's domain is not compatible with `bool` (in the case where
 3049 `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3050 Description

3051 GrB_mxm computes the matrix product $C = A \oplus . \otimes B$ or, if an optional binary accumulation operator
 3052 (\odot) is provided, $C = C \odot (A \oplus . \otimes B)$ (where matrices A and B can be optionally transposed).
 3053 Logically, this operation occurs in three steps:

3054 **Setup** The internal matrices and mask used in the computation are formed and their domains
 3055 and dimensions are tested for compatibility.

3056 **Compute** The indicated computations are carried out.

3057 **Output** The result is written into the output matrix, possibly under control of a mask.

3058 Up to four argument matrices are used in the GrB_mxm operation:

- 3059 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3060 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)
- 3061 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3062 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3063 The argument matrices, the semiring, and the accumulation operator (if provided) are tested for
 3064 domain compatibility as follows:

- 3065 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 3066 must be from one of the pre-defined types of Table 3.2.
- 3067 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the semiring.
- 3068 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the semiring.
- 3069 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the semiring.
- 3070 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3071 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
 3072 of the accumulation operator.

3073 Two domains are compatible with each other if values from one domain can be cast to values in
 3074 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are
 3075 all compatible with each other. A domain from a user-defined type is only compatible with itself.

3076 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the domain mismatch
3077 error listed above is returned.

3078 From the argument matrices, the internal matrices and mask used in the computation are formed
3079 (\leftarrow denotes copy):

- 3080 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 3081 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 3082 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
3083 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 3084 (b) If `Mask \neq GrB_NULL`,
 - 3085 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
3086 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 3087 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
3088 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 3089 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 3090 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3091 4. Matrix $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

3092 The internal matrices and masks are checked for dimension compatibility. The following conditions
3093 must hold:

- 3094 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 3095 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 3096 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
- 3097 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.
- 3098 5. $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.

3099 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the dimension mismatch
3100 error listed above is returned.

3101 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
3102 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3103 We are now ready to carry out the matrix multiplication and any additional associated operations.
3104 We describe this in terms of two intermediate matrices:

- 3105 • $\tilde{\mathbf{T}}$: The matrix holding the product of matrices $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 3106 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

3107 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

$$3109 \quad T_{ij} = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{B}}(:, j))} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{B}}(k, j)),$$

3110 where \oplus and \otimes are the additive and multiplicative operators of semiring \mathbf{op} , respectively.

3111 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 3112 • If $\mathbf{accum} = \mathbf{GrB_NULL}$, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 3113 • If \mathbf{accum} is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$3114 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3115 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
3116 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 3117 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 3118 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3119 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3120 \quad & \\ 3121 \end{aligned}$$

3122 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

3123 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
3124 using what is called a *standard matrix mask and replace*. This is carried out under control of the
3125 mask which acts as a “write mask”.

- 3126 • If $\mathbf{desc}[\mathbf{GrB_OUTP}].\mathbf{GrB_REPLACE}$ is set, then any values in \mathbf{C} on input to this operation are
3127 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$3128 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3129 • If $\mathbf{desc}[\mathbf{GrB_OUTP}].\mathbf{GrB_REPLACE}$ is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
3130 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
3131 mask are unchanged:

$$3132 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3133 In $\mathbf{GrB_BLOCKING}$ mode, the method exits with return value $\mathbf{GrB_SUCCESS}$ and the new content
3134 of matrix \mathbf{C} is as defined above and fully computed. In $\mathbf{GrB_NONBLOCKING}$ mode, the method
3135 exits with return value $\mathbf{GrB_SUCCESS}$ and the new content of matrix \mathbf{C} is as defined above but
3136 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
3137 sequence.

3138 4.3.2 vxm: Vector-matrix multiply

3139 Multiplies a (row) vector with a matrix on an semiring. The result is a vector.

3140 C Syntax

```
3141      GrB_Info GrB_vxm(GrB_Vector      w,  
3142                      const GrB_Vector mask,  
3143                      const GrB_BinaryOp accum,  
3144                      const GrB_Semiring op,  
3145                      const GrB_Vector u,  
3146                      const GrB_Matrix A,  
3147                      const GrB_Descriptor desc);
```

3148 Parameters

3149 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3150 that may be accumulated with the result of the vector-matrix product. On output,
3151 this vector holds the results of the operation.

3152 **mask** (IN) An optional “write” mask that controls which results from this operation are
3153 stored into the output vector **w**. The mask dimensions must match those of the
3154 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
3155 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
3156 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
3157 dimensions of **w**), **GrB_NULL** should be specified.

3158 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
3159 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
3160 specified.

3161 **op** (IN) Semiring used in the vector-matrix multiply.

3162 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the
3163 multiplication.

3164 **A** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
3165 multiplication.

3166 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
3167 should be specified. Non-default field/value pairs are listed as follows:
3168

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.
A	GrB_INP1	GrB_TRAN	Use transpose of A for the operation.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector w is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

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

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to `new` (or `dup` for matrix or vector parameters).

GrB_DIMENSION_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

GrB_DOMAIN_MISMATCH The domains of the various vectors/matrices are incompatible with the corresponding domains of the semiring or accumulation operator, or the mask's domain is not compatible with `bool` (in the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

Description

GrB_vxm computes the vector-matrix product $w^T = u^T \oplus . \otimes A$, or, if an optional binary accumulation operator (\odot) is provided, $w^T = w^T \odot (u^T \oplus . \otimes A)$ (where matrix A can be optionally transposed). Logically, this operation occurs in three steps:

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

Compute The indicated computations are carried out.

3196 **Output** The result is written into the output vector, possibly under control of a mask.

3197 Up to four argument vectors or matrices are used in the `GrB_vxm` operation:

- 3198 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3199 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3200 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3201 4. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

3202 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are
 3203 tested for domain compatibility as follows:

- 3204 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
 3205 must be from one of the pre-defined types of Table 3.2.
- 3206 2. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the semiring.
- 3207 3. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$ of the semiring.
- 3208 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$ of the semiring.
- 3209 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
 3210 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$
 3211 of the accumulation operator.

3212 Two domains are compatible with each other if values from one domain can be cast to values in
 3213 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are
 3214 all compatible with each other. A domain from a user-defined type is only compatible with itself.
 3215 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the domain mismatch
 3216 error listed above is returned.

3217 From the argument vectors and matrices, the internal matrices and mask used in the computation
 3218 are formed (\leftarrow denotes copy):

- 3219 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3220 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3221 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 3222 (b) If `mask` \neq `GrB_NULL`,
 - 3223 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 3224 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}(\mathbf{mask})(i) = \mathbf{true})\} \rangle$.
 - 3225 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3226 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

3227 4. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP1}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

3228 The internal matrices and masks are checked for shape compatibility. The following conditions
3229 must hold:

3230 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$.

3231 2. $\text{size}(\tilde{\mathbf{w}}) = \text{ncols}(\tilde{\mathbf{A}})$.

3232 3. $\text{size}(\tilde{\mathbf{u}}) = \text{nrows}(\tilde{\mathbf{A}})$.

3233 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the dimension mismatch
3234 error listed above is returned.

3235 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
3236 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3237 We are now ready to carry out the vector-matrix multiplication and any additional associated
3238 operations. We describe this in terms of two intermediate vectors:

- 3239 • $\tilde{\mathbf{t}}$: The vector holding the product of vector $\tilde{\mathbf{u}}^T$ and matrix $\tilde{\mathbf{A}}$.
- 3240 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3241 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.
3242 The value of each of its elements is computed by

$$3243 \quad t_j = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{A}}(:, j))} (\tilde{\mathbf{u}}(k) \otimes \tilde{\mathbf{A}}(k, j)),$$

3244 where \oplus and \otimes are the additive and multiplicative operators of semiring `op`, respectively.

3245 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 3246 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 3247 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3248 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3249 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
3250 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 3251 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 3252 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3253 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3254 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3255 \end{aligned}$$

3256 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3257 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 3258 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3259 mask which acts as a “write mask”.

- 3260 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{w} on input to this operation are
 3261 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$3262 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3263 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3264 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 3265 mask are unchanged:

$$3266 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3267 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 3268 of vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 3269 exits with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but
 3270 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3271 sequence.

3272 4.3.3 mxv: Matrix-vector multiply

3273 Multiplies a matrix by a vector on a semiring. The result is a vector.

3274 C Syntax

```
3275      GrB_Info GrB_mxv(GrB_Vector      w,
3276                      const GrB_Vector mask,
3277                      const GrB_BinaryOp accum,
3278                      const GrB_Semiring op,
3279                      const GrB_Matrix A,
3280                      const GrB_Vector u,
3281                      const GrB_Descriptor desc);
```

3282 Parameters

3283 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
 3284 that may be accumulated with the result of the matrix-vector product. On output,
 3285 this vector holds the results of the operation.

3286 **mask** (IN) An optional “write” mask that controls which results from this operation are
 3287 stored into the output vector \mathbf{w} . The mask dimensions must match those of the
 3288 vector \mathbf{w} . If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain

3289 of the `mask` vector must be of type `bool` or any of the predefined “built-in” types
 3290 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
 3291 dimensions of `w`), `GrB_NULL` should be specified.

3292 `accum` (IN) An optional binary operator used for accumulating entries into existing `w`
 3293 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
 3294 specified.

3295 `op` (IN) Semiring used in the vector-matrix multiply.

3296 `A` (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 3297 multiplication.

3298 `u` (IN) The GraphBLAS vector holding the values for the right-hand vector in the
 3299 multiplication.

3300 `desc` (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
 3301 should be specified. Non-default field/value pairs are listed as follows:
 3302

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.

3304 Return Values

3305 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
 3306 blocking mode, this indicates that the compatibility tests on di-
 3307 mensions and domains for the input arguments passed successfully.
 3308 Either way, output vector `w` is ready to be used in the next method
 3309 of the sequence.

3310 `GrB_PANIC` Unknown internal error.

3311 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
 3312 GraphBLAS objects (input or output) is in an invalid state caused
 3313 by a previous execution error. Call `GrB_error()` to access any error
 3314 messages generated by the implementation.

3315 `GrB_OUT_OF_MEMORY` Not enough memory available for the operation.

3316 `GrB_UNINITIALIZED_OBJECT` One or more of the GraphBLAS objects has not been initialized by
 3317 a call to `new` (or `dup` for matrix or vector parameters).

3318 GrB_DIMENSION_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

3319 GrB_DOMAIN_MISMATCH The domains of the various vectors/matrices are incompatible with
3320 the corresponding domains of the semiring or accumulation opera-
3321 tor, or the mask's domain is not compatible with **bool** (in the case
3322 where desc[GrB_MASK].GrB_STRUCTURE is not set).

3323 Description

3324 GrB_mxv computes the matrix-vector product $\mathbf{w} = \mathbf{A} \oplus . \otimes \mathbf{u}$, or, if an optional binary accumulation
3325 operator (\odot) is provided, $\mathbf{w} = \mathbf{w} \odot (\mathbf{A} \oplus . \otimes \mathbf{u})$ (where matrix \mathbf{A} can be optionally transposed).
3326 Logically, this operation occurs in three steps:

3327 **Setup** The internal vectors, matrices and mask used in the computation are formed and their
3328 domains/dimensions are tested for compatibility.

3329 **Compute** The indicated computations are carried out.

3330 **Output** The result is written into the output vector, possibly under control of a mask.

3331 Up to four argument vectors or matrices are used in the GrB_mxv operation:

- 3332 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3333 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3334 3. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$
- 3335 4. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

3336 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are
3337 tested for domain compatibility as follows:

- 3338 1. If **mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\mathbf{mask})$
3339 must be from one of the pre-defined types of Table 3.2.
- 3340 2. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the semiring.
- 3341 3. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$ of the semiring.
- 3342 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$ of the semiring.
- 3343 5. If **accum** is not GrB_NULL, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
3344 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$
3345 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

$$t_i = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{A}}(i, :)) \cap \text{ind}(\tilde{\mathbf{u}})} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{u}}(k)),$$

3378 where \oplus and \otimes are the additive and multiplicative operators of semiring **op**, respectively.
 3379 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 3380 • If **accum** = **GrB_NULL**, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 3381 • If **accum** is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3382 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3383 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 3384 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 3385 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 3386 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3387 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3388 \end{aligned}$$

3389 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

3391 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector **w**,
 3392 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3393 mask which acts as a “write mask”.

- 3394 • If **desc[GrB_OUTP].GrB_REPLACE** is set, then any values in **w** on input to this operation are
 3395 deleted and the content of the new output vector, **w**, is defined as,

$$3396 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3397 • If **desc[GrB_OUTP].GrB_REPLACE** is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3398 copied into the result vector, **w**, and elements of **w** that fall outside the set indicated by the
 3399 mask are unchanged:

$$3400 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3401 In **GrB_BLOCKING** mode, the method exits with return value **GrB_SUCCESS** and the new content
 3402 of vector **w** is as defined above and fully computed. In **GrB_NONBLOCKING** mode, the method
 3403 exits with return value **GrB_SUCCESS** and the new content of vector **w** is as defined above but
 3404 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3405 sequence.

3406 4.3.4 eWiseMult: Element-wise multiplication

3407 **Note:** The difference between **eWiseAdd** and **eWiseMult** is not about the element-wise operation
 3408 but how the index sets are treated. **eWiseAdd** returns an object whose indices are the “union” of
 3409 the indices of the inputs whereas **eWiseMult** returns an object whose indices are the “intersection”
 3410 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on
 3411 the set of values from the resulting index set.

3412 4.3.4.1 eWiseMult: Vector variant

3413 Perform element-wise (general) multiplication on the intersection of elements of two vectors, pro-
3414 ducing a third vector as result.

3415 C Syntax

```
3416     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3417                           const GrB_Vector mask,  
3418                           const GrB_BinaryOp accum,  
3419                           const GrB_Semiring op,  
3420                           const GrB_Vector u,  
3421                           const GrB_Vector v,  
3422                           const GrB_Descriptor desc);  
3423  
3424     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3425                           const GrB_Vector mask,  
3426                           const GrB_BinaryOp accum,  
3427                           const GrB_Monoid op,  
3428                           const GrB_Vector u,  
3429                           const GrB_Vector v,  
3430                           const GrB_Descriptor desc);  
3431  
3432     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3433                           const GrB_Vector mask,  
3434                           const GrB_BinaryOp accum,  
3435                           const GrB_BinaryOp op,  
3436                           const GrB_Vector u,  
3437                           const GrB_Vector v,  
3438                           const GrB_Descriptor desc);
```

3439 Parameters

3440 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3441 that may be accumulated with the result of the element-wise operation. On output,
3442 this vector holds the results of the operation.

3443 **mask** (IN) An optional “write” mask that controls which results from this operation are
3444 stored into the output vector **w**. The mask dimensions must match those of the
3445 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
3446 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
3447 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
3448 dimensions of **w**), **GrB_NULL** should be specified.

3449 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**

3450 entries. If assignment rather than accumulation is desired, GrB_NULL should be
 3451 specified.

3452 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
 3453 operation. Depending on which type is passed, the following defines the binary
 3454 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

3455 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

3456 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-
 3457 nored.

3458 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
 3459 is ignored.

3460 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the
 3461 operation.

3462 **v** (IN) The GraphBLAS vector holding the values for the right-hand vector in the
 3463 operation.

3464 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 3465 should be specified. Non-default field/value pairs are listed as follows:
 3466

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.

3468 Return Values

3469 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 3470 blocking mode, this indicates that the compatibility tests on di-
 3471 mensions and domains for the input arguments passed successfully.
 3472 Either way, output vector w is ready to be used in the next method
 3473 of the sequence.

3474 GrB_PANIC Unknown internal error.

3475 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 3476 GraphBLAS objects (input or output) is in an invalid state caused
 3477 by a previous execution error. Call GrB_error() to access any error
 3478 messages generated by the implementation.

3479 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3480 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 3481 a call to `new` (or `dup` for vector parameters).

3482 GrB_DIMENSION_MISMATCH Mask or vector dimensions are incompatible.

3483 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
 3484 responding domains of the binary operator (`op`) or accumulation
 3485 operator, or the mask's domain is not compatible with `bool` (in the
 3486 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3487 Description

3488 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS vectors:
 3489 $\mathbf{w} = \mathbf{u} \otimes \mathbf{v}$, or, if an optional binary accumulation operator (\odot) is provided, $\mathbf{w} = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$.
 3490 Logically, this operation occurs in three steps:

3491 **Setup** The internal vectors and mask used in the computation are formed and their domains
 3492 and dimensions are tested for compatibility.

3493 **Compute** The indicated computations are carried out.

3494 **Output** The result is written into the output vector, possibly under control of a mask.

3495 Up to four argument vectors are used in the `GrB_eWiseMult` operation:

- 3496 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3497 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3498 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3499 4. $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

3500 The argument vectors, the “product” operator (`op`), and the accumulation operator (if provided)
 3501 are tested for domain compatibility as follows:

- 3502 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
 3503 must be from one of the pre-defined types of Table 3.2.
- 3504 2. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$.
- 3505 3. $\mathbf{D}(\mathbf{v})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$.
- 3506 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$.
- 3507 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
 3508 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of
 3509 the accumulation operator.

3510 Two domains are compatible with each other if values from one domain can be cast to values in
 3511 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 3512 compatible with each other. A domain from a user-defined type is only compatible with itself. If any
 3513 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch
 3514 error listed above is returned.

3515 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 3516 denotes copy):

- 3517 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3518 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3519 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 3520 (b) If `mask \neq GrB_NULL`,
 - 3521 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 3522 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 3523 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3524 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 3525 4. Vector $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$.

3526 The internal vectors and mask are checked for dimension compatibility. The following conditions
 3527 must hold:

- 3528 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}}) = \mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{v}})$.

3529 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension
 3530 mismatch error listed above is returned.

3531 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3532 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3533 We are now ready to carry out the element-wise “product” and any additional associated operations.
 3534 We describe this in terms of two intermediate vectors:

- 3535 • $\tilde{\mathbf{t}}$: The vector holding the element-wise “product” of $\tilde{\mathbf{u}}$ and vector $\tilde{\mathbf{v}}$.
- 3536 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3537 The intermediate vector $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, t_i) : \mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}}) \neq \emptyset\} \rangle$ is created. The
 3538 value of each of its elements is computed by:

$$3539 \quad t_i = (\tilde{\mathbf{u}}(i) \otimes \tilde{\mathbf{v}}(i)), \forall i \in (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}}))$$

3540 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

3541 • If $\text{accum} = \text{GrB_NULL}$, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

3542 • If accum is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

3543
$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3544 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 3545 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

3546
$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

3547

3548
$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3549

3550
$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3551 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3552 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 3553 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3554 mask which acts as a “write mask”.

3555 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is set, then any values in \mathbf{w} on input to this operation are
 3556 deleted and the content of the new output vector, \mathbf{w} , is defined as,

3557
$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3558 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3559 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 3560 mask are unchanged:

3561
$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3562 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 3563 of vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 3564 exits with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but
 3565 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3566 sequence.

3567 4.3.4.2 eWiseMult: Matrix variant

3568 Perform element-wise (general) multiplication on the intersection of elements of two matrices, pro-
 3569 ducing a third matrix as result.

3570 C Syntax

```

3571     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3572                           const GrB_Matrix Mask,
3573                           const GrB_BinaryOp accum,
3574                           const GrB_Semiring op,
3575                           const GrB_Matrix A,
3576                           const GrB_Matrix B,
3577                           const GrB_Descriptor desc);
3578
3579     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3580                           const GrB_Matrix Mask,
3581                           const GrB_BinaryOp accum,
3582                           const GrB_Monoid op,
3583                           const GrB_Matrix A,
3584                           const GrB_Matrix B,
3585                           const GrB_Descriptor desc);
3586
3587     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3588                           const GrB_Matrix Mask,
3589                           const GrB_BinaryOp accum,
3590                           const GrB_BinaryOp op,
3591                           const GrB_Matrix A,
3592                           const GrB_Matrix B,
3593                           const GrB_Descriptor desc);

```

3594 Parameters

3595 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
3596 that may be accumulated with the result of the element-wise operation. On output,
3597 the matrix holds the results of the operation.

3598 **Mask** (IN) An optional “write” mask that controls which results from this operation are
3599 stored into the output matrix C. The mask dimensions must match those of the
3600 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
3601 of the `Mask` matrix must be of type `bool` or any of the predefined “built-in” types
3602 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
3603 dimensions of C), `GrB_NULL` should be specified.

3604 **accum** (IN) An optional binary operator used for accumulating entries into existing C
3605 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
3606 specified.

3607 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
3608 operation. Depending on which type is passed, the following defines the binary
3609 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

3610 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.
 3611 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-
 3612 nored.
 3613 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
 3614 is ignored.

3615 A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 3616 operation.

3617 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
 3618 operation.

3619 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 3620 should be specified. Non-default field/value pairs are listed as follows:
 3621

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.

3623 Return Values

3624 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 3625 blocking mode, this indicates that the compatibility tests on di-
 3626 mensions and domains for the input arguments passed successfully.
 3627 Either way, output matrix C is ready to be used in the next method
 3628 of the sequence.

3629 GrB_PANIC Unknown internal error.

3630 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 3631 GraphBLAS objects (input or output) is in an invalid state caused
 3632 by a previous execution error. Call GrB_error() to access any error
 3633 messages generated by the implementation.

3634 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3635 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 3636 a call to new (or Matrix_dup for matrix parameters).

3637 GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

3638 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 3639 corresponding domains of the binary operator (op) or accumulation
 3640 operator, or the mask's domain is not compatible with `bool` (in the
 3641 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3642 Description

3643 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS matrices:
 3644 $C = A \otimes B$, or, if an optional binary accumulation operator (\odot) is provided, $C = C \odot (A \otimes B)$.
 3645 Logically, this operation occurs in three steps:

3646 **Setup** The internal matrices and mask used in the computation are formed and their domains
 3647 and dimensions are tested for compatibility.

3648 **Compute** The indicated computations are carried out.

3649 **Output** The result is written into the output matrix, possibly under control of a mask.

3650 Up to four argument matrices are used in the `GrB_eWiseMult` operation:

- 3651 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3652 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)
- 3653 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3654 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3655 The argument matrices, the “product” operator (op), and the accumulation operator (if provided)
 3656 are tested for domain compatibility as follows:

- 3657 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 3658 must be from one of the pre-defined types of Table 3.2.
- 3659 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 3660 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 3661 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3662 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3663 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of op must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
 3664 the accumulation operator.

3665 Two domains are compatible with each other if values from one domain can be cast to values in
 3666 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 3667 compatible with each other. A domain from a user-defined type is only compatible with itself. If any

3668 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch
 3669 error listed above is returned.

3670 From the argument matrices, the internal matrices and mask used in the computation are formed
 3671 (\leftarrow denotes copy):

- 3672 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 3673 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 3674 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 3675 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 3676 (b) If `Mask \neq GrB_NULL`,
 - 3677 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 3678 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 3679 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 3680 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\text{bool})\mathbf{Mask}(i, j) = \text{true}\} \rangle$.
 - 3681 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 3682 3. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3683 4. Matrix $\tilde{\mathbf{B}} \leftarrow \text{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

3684 The internal matrices and masks are checked for dimension compatibility. The following conditions
 3685 must hold:

- 3686 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.
- 3687 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.

3688 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension
 3689 mismatch error listed above is returned.

3690 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3691 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3692 We are now ready to carry out the element-wise “product” and any additional associated operations.
 3693 We describe this in terms of two intermediate matrices:

- 3694 • $\tilde{\mathbf{T}}$: The matrix holding the element-wise product of $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 3695 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

3696 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\} \rangle$
 3697 is created. The value of each of its elements is computed by

$$3698 \quad T_{ij} = (\tilde{\mathbf{A}}(i, j) \otimes \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})$$

3699 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

3700 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.

3701 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$3702 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3703 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
3704 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$3705 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$3706 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$3707 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3710 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3711 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
3712 using what is called a *standard matrix mask and replace*. This is carried out under control of the
3713 mask which acts as a “write mask”.

3714 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
3715 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$3716 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3717 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
3718 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
3719 mask are unchanged:

$$3720 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3721 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
3722 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
3723 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
3724 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
3725 sequence.

3726 4.3.5 eWiseAdd: Element-wise addition

3727 **Note:** The difference between `eWiseAdd` and `eWiseMult` is not about the element-wise operation
3728 but how the index sets are treated. `eWiseAdd` returns an object whose indices are the “union” of
3729 the indices of the inputs whereas `eWiseMult` returns an object whose indices are the “intersection”
3730 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on
3731 the set of values from the resulting index set.

3732 4.3.5.1 eWiseAdd: Vector variant

3733 Perform element-wise (general) addition on the elements of two vectors, producing a third vector
3734 as result.

3735 C Syntax

```
3736     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3737                           const GrB_Vector mask,  
3738                           const GrB_BinaryOp accum,  
3739                           const GrB_Semiring op,  
3740                           const GrB_Vector u,  
3741                           const GrB_Vector v,  
3742                           const GrB_Descriptor desc);  
3743  
3744     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3745                           const GrB_Vector mask,  
3746                           const GrB_BinaryOp accum,  
3747                           const GrB_Monoid op,  
3748                           const GrB_Vector u,  
3749                           const GrB_Vector v,  
3750                           const GrB_Descriptor desc);  
3751  
3752     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3753                           const GrB_Vector mask,  
3754                           const GrB_BinaryOp accum,  
3755                           const GrB_BinaryOp op,  
3756                           const GrB_Vector u,  
3757                           const GrB_Vector v,  
3758                           const GrB_Descriptor desc);
```

3759 Parameters

3760 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3761 that may be accumulated with the result of the element-wise operation. On output,
3762 this vector holds the results of the operation.

3763 **mask** (IN) An optional “write” mask that controls which results from this operation are
3764 stored into the output vector **w**. The mask dimensions must match those of the
3765 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
3766 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
3767 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
3768 dimensions of **w**), **GrB_NULL** should be specified.

3769 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**

3770 entries. If assignment rather than accumulation is desired, GrB_NULL should be
3771 specified.

3772 op (IN) The semiring, monoid, or binary operator used in the element-wise “sum”
3773 operation. Depending on which type is passed, the following defines the binary
3774 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$, used:

3775 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

3776 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-
3777 nored.

3778 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-
3779 nary op and additive identity are ignored.

3780 u (IN) The GraphBLAS vector holding the values for the left-hand vector in the
3781 operation.

3782 v (IN) The GraphBLAS vector holding the values for the right-hand vector in the
3783 operation.

3784 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
3785 should be specified. Non-default field/value pairs are listed as follows:
3786

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.

3788 Return Values

3789 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
3790 blocking mode, this indicates that the compatibility tests on di-
3791 mensions and domains for the input arguments passed successfully.
3792 Either way, output vector w is ready to be used in the next method
3793 of the sequence.

3794 GrB_PANIC Unknown internal error.

3795 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
3796 GraphBLAS objects (input or output) is in an invalid state caused
3797 by a previous execution error. Call GrB_error() to access any error
3798 messages generated by the implementation.

3799 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3800 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
3801 a call to `new` (or `dup` for vector parameters).

3802 GrB_DIMENSION_MISMATCH Mask or vector dimensions are incompatible.

3803 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
3804 responding domains of the binary operator (`op`) or accumulation
3805 operator, or the mask's domain is not compatible with `bool` (in the
3806 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3807 Description

3808 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS vectors: $\mathbf{w} =$
3809 $\mathbf{u} \oplus \mathbf{v}$, or, if an optional binary accumulation operator (\odot) is provided, $\mathbf{w} = \mathbf{w} \odot (\mathbf{u} \oplus \mathbf{v})$. Logically,
3810 this operation occurs in three steps:

3811 **Setup** The internal vectors and mask used in the computation are formed and their domains
3812 and dimensions are tested for compatibility.

3813 **Compute** The indicated computations are carried out.

3814 **Output** The result is written into the output vector, possibly under control of a mask.

3815 Up to four argument vectors are used in the `GrB_eWiseAdd` operation:

- 3816 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3817 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3818 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3819 4. $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

3820 The argument vectors, the “sum” operator (`op`), and the accumulation operator (if provided) are
3821 tested for domain compatibility as follows:

- 3822 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
3823 must be from one of the pre-defined types of Table 3.2.
- 3824 2. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$.
- 3825 3. $\mathbf{D}(\mathbf{v})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$.
- 3826 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$.
- 3827 5. $\mathbf{D}(\mathbf{u})$ and $\mathbf{D}(\mathbf{v})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$.
- 3828 6. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
3829 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of
3830 the accumulation operator.

3831 Two domains are compatible with each other if values from one domain can be cast to values in
 3832 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 3833 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 3834 any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the domain mismatch
 3835 error listed above is returned.

3836 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 3837 denotes copy):

- 3838 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3839 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3840 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
 - 3841 (b) If `mask \neq GrB_NULL`,
 - 3842 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 - 3843 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 3844 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3845 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 3846 4. Vector $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$.

3847 The internal vectors and mask are checked for dimension compatibility. The following conditions
 3848 must hold:

- 3849 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}}) = \text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{v}})$.

3850 If any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the dimension
 3851 mismatch error listed above is returned.

3852 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3853 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3854 We are now ready to carry out the element-wise “sum” and any additional associated operations.
 3855 We describe this in terms of two intermediate vectors:

- 3856 • $\tilde{\mathbf{t}}$: The vector holding the element-wise “sum” of $\tilde{\mathbf{u}}$ and vector $\tilde{\mathbf{v}}$.
- 3857 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3858 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
 3859 value of each of its elements is computed by:

$$\begin{aligned}
 3860 \quad t_i &= (\tilde{\mathbf{u}}(i) \oplus \tilde{\mathbf{v}}(i)), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}})) \\
 3861 \\
 3862 \quad t_i &= \tilde{\mathbf{u}}(i), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) - (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}})))
 \end{aligned}$$

3863
3864

$$t_i = \tilde{\mathbf{v}}(i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{v}}) - (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}})))$$

3865 where the difference operator in the previous expressions refers to set difference.

3866 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 3867 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 3868 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3870 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
3871 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 3872 z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 3873 z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3874 z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3875 \end{aligned}$$

3876 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3878 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
3879 using what is called a *standard vector mask and replace*. This is carried out under control of the
3880 mask which acts as a “write mask”.

- 3881 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
3882 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3884 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
3885 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
3886 mask are unchanged:

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3888 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
3889 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
3890 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
3891 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
3892 sequence.

3893 4.3.5.2 eWiseAdd: Matrix variant

3894 Perform element-wise (general) addition on the elements of two matrices, producing a third matrix
3895 as result.

3896 C Syntax

```

3897     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3898                           const GrB_Matrix Mask,
3899                           const GrB_BinaryOp accum,
3900                           const GrB_Semiring op,
3901                           const GrB_Matrix A,
3902                           const GrB_Matrix B,
3903                           const GrB_Descriptor desc);
3904
3905     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3906                           const GrB_Matrix Mask,
3907                           const GrB_BinaryOp accum,
3908                           const GrB_Monoid op,
3909                           const GrB_Matrix A,
3910                           const GrB_Matrix B,
3911                           const GrB_Descriptor desc);
3912
3913     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3914                           const GrB_Matrix Mask,
3915                           const GrB_BinaryOp accum,
3916                           const GrB_BinaryOp op,
3917                           const GrB_Matrix A,
3918                           const GrB_Matrix B,
3919                           const GrB_Descriptor desc);

```

3920 Parameters

3921 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
3922 that may be accumulated with the result of the element-wise operation. On output,
3923 the matrix holds the results of the operation.

3924 **Mask** (IN) An optional “write” mask that controls which results from this operation are
3925 stored into the output matrix C. The mask dimensions must match those of the
3926 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
3927 of the `Mask` matrix must be of type `bool` or any of the predefined “built-in” types
3928 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
3929 dimensions of C), `GrB_NULL` should be specified.

3930 **accum** (IN) An optional binary operator used for accumulating entries into existing C
3931 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
3932 specified.

3933 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”
3934 operation. Depending on which type is passed, the following defines the binary
3935 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$, used:

3936 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.
 3937 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-
 3938 nored.
 3939 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-
 3940 nary op and additive identity are ignored.

3941 A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 3942 operation.

3943 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
 3944 operation.

3945 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 3946 should be specified. Non-default field/value pairs are listed as follows:
 3947

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.

3949 Return Values

3950 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 3951 blocking mode, this indicates that the compatibility tests on di-
 3952 mensions and domains for the input arguments passed successfully.
 3953 Either way, output matrix C is ready to be used in the next method
 3954 of the sequence.

3955 GrB_PANIC Unknown internal error.

3956 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 3957 GraphBLAS objects (input or output) is in an invalid state caused
 3958 by a previous execution error. Call GrB_error() to access any error
 3959 messages generated by the implementation.

3960 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3961 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 3962 a call to new (or Matrix_dup for matrix parameters).

3963 GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

3964 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 3965 corresponding domains of the binary operator (op) or accumulation
 3966 operator, or the mask's domain is not compatible with `bool` (in the
 3967 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3968 Description

3969 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS matrices:
 3970 $C = A \oplus B$, or, if an optional binary accumulation operator (\odot) is provided, $C = C \odot (A \oplus B)$.
 3971 Logically, this operation occurs in three steps:

3972 **Setup** The internal matrices and mask used in the computation are formed and their domains
 3973 and dimensions are tested for compatibility.

3974 **Compute** The indicated computations are carried out.

3975 **Output** The result is written into the output matrix, possibly under control of a mask.

3976 Up to four argument matrices are used in the `GrB_eWiseAdd` operation:

- 3977 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3978 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)
- 3979 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3980 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3981 The argument matrices, the “sum” operator (op), and the accumulation operator (if provided) are
 3982 tested for domain compatibility as follows:

- 3983 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 3984 must be from one of the pre-defined types of Table 3.2.
- 3985 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 3986 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 3987 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3988 5. $\mathbf{D}(A)$ and $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3989 6. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3990 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
 3991 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.

4023 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cup \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\}$
 4024 is created. The value of each of its elements is computed by

$$\begin{aligned}
 4025 \quad T_{ij} &= (\tilde{\mathbf{A}}(i, j) \oplus \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \\
 4026 \quad T_{ij} &= \tilde{\mathbf{A}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{A}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}))) \\
 4027 \quad T_{ij} &= \tilde{\mathbf{B}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{B}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})))
 \end{aligned}$$

4030 where the difference operator in the previous expressions refers to set difference.

4031 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 4032 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 4033 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$4034 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4035 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 4036 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned}
 4037 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\
 4038 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 4039 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 4040 \quad & \\
 4041 \quad &
 \end{aligned}$$

4042 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4043 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 4044 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 4045 mask which acts as a “write mask”.

- 4046 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
 4047 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$4048 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 4049 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 4050 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 4051 mask are unchanged:

$$4052 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4053 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 4054 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 4055 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
 4056 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 4057 sequence.

4058 4.3.6 extract: Selecting sub-graphs

4059 Extract a subset of a matrix or vector.

4060 4.3.6.1 extract: Standard vector variant

4061 Extract a sub-vector from a larger vector as specified by a set of indices. The result is a vector
4062 whose size is equal to the number of indices.

4063 C Syntax

```
4064         GrB_Info GrB_extract(GrB_Vector          w,  
4065                             const GrB_Vector    mask,  
4066                             const GrB_BinaryOp   accum,  
4067                             const GrB_Vector    u,  
4068                             const GrB_Index     *indices,  
4069                             GrB_Index           nindices,  
4070                             const GrB_Descriptor desc);
```

4071 Parameters

4072 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4073 that may be accumulated with the result of the extract operation. On output, this
4074 vector holds the results of the operation.

4075 **mask** (IN) An optional “write” mask that controls which results from this operation are
4076 stored into the output vector **w**. The mask dimensions must match those of the
4077 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4078 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
4079 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
4080 dimensions of **w**), **GrB_NULL** should be specified.

4081 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
4082 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4083 specified.

4084 **u** (IN) The GraphBLAS vector from which the subset is extracted.

4085 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations of
4086 elements from **u** that are extracted. If all elements of **u** are to be extracted in order
4087 from 0 to **nindices** – 1, then **GrB_ALL** should be specified. Regardless of execution
4088 mode and return value, this array may be manipulated by the caller after this
4089 operation returns without affecting any deferred computations for this operation.

4090 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(w)**.

4091 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 4092 should be specified. Non-default field/value pairs are listed as follows:
 4093

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 .

4095 Return Values

4096 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 4097 blocking mode, this indicates that the compatibility tests on
 4098 dimensions and domains for the input arguments passed suc-
 4099 cessfully. Either way, output vector **w** is ready to be used in the
 4100 next method of the sequence.

4101 GrB_PANIC Unknown internal error.

4102 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
 4103 opaque GraphBLAS objects (input or output) is in an invalid
 4104 state caused by a previous execution error. Call GrB_error() to
 4105 access any error messages generated by the implementation.

4106 GrB_OUT_OF_MEMORY Not enough memory available for operation.

4107 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
 4108 by a call to **new** (or **dup** for vector parameters).

4109 GrB_INDEX_OUT_OF_BOUNDS A value in **indices** is greater than or equal to **size(u)**. In non-
 4110 blocking mode, this error can be deferred.

4111 GrB_DIMENSION_MISMATCH **mask** and **w** dimensions are incompatible, or **nindices** \neq **size(w)**.

4112 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with each
 4113 other or the corresponding domains of the accumulation oper-
 4114 ator, or the mask's domain is not compatible with **bool** (in the
 4115 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

4116 GrB_NULL_POINTER Argument **row_indices** is a NULL pointer.

4117 Description

4118 This variant of GrB_extract computes the result of extracting a subset of locations from a Graph-
 4119 BLAS vector in a specific order: **w** = **u(indices)**; or, if an optional binary accumulation operator

4120 (\odot) is provided, $w = w \odot u(\text{indices})$. More explicitly:

$$4121 \quad \begin{aligned} w(i) &= u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ w(i) &= w(i) \odot u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices} \end{aligned}$$

4122 Logically, this operation occurs in three steps:

4123 **Setup** The internal vectors and mask used in the computation are formed and their domains
4124 and dimensions are tested for compatibility.

4125 **Compute** The indicated computations are carried out.

4126 **Output** The result is written into the output vector, possibly under control of a mask.

4127 Up to three argument vectors are used in this `GrB_extract` operation:

- 4128 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4129 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4130 3. $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

4131 The argument vectors and the accumulation operator (if provided) are tested for domain compati-
4132 bility as follows:

- 4133 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
4134 must be from one of the pre-defined types of Table 3.2.
- 4135 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}(u)$.
- 4136 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4137 of the accumulation operator and $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
4138 mulation operator.

4139 Two domains are compatible with each other if values from one domain can be cast to values in
4140 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4141 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4142 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch
4143 error listed above is returned.

4144 From the arguments, the internal vectors, mask, and index array used in the computation are
4145 formed (\leftarrow denotes copy):

- 4146 1. Vector $\tilde{w} \leftarrow w$.
- 4147 2. One-dimensional mask, \tilde{m} , is computed from argument `mask` as follows:
4148 (a) If `mask` = `GrB_NULL`, then $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$.

- 4149 (b) If $\text{mask} \neq \text{GrB_NULL}$,
- 4150 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
- 4151 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
- 4152 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
- 4153 3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 4154 4. The internal index array, $\widetilde{\mathbf{I}}$, is computed from argument indices as follows:
- 4155 (a) If $\text{indices} = \text{GrB_ALL}$, then $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$.
- 4156 (b) Otherwise, $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$.

4157 The internal vectors and mask are checked for dimension compatibility. The following conditions
4158 must hold:

- 4159 1. $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 4160 2. $\text{nindices} = \text{size}(\widetilde{\mathbf{w}})$.

4161 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
4162 match error listed above is returned.

4163 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4164 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4165 We are now ready to carry out the extract and any additional associated operations. We describe
4166 this in terms of two intermediate vectors:

- 4167 • $\widetilde{\mathbf{t}}$: The vector holding the extraction from $\widetilde{\mathbf{u}}$ in their destination locations relative to $\widetilde{\mathbf{w}}$.
- 4168 • $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4169 The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$4170 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{w}}), \{(i, \widetilde{\mathbf{u}}[\widetilde{\mathbf{I}}[i]]) \mid \forall i, 0 \leq i < \text{nindices} : \widetilde{\mathbf{I}}[i] \in \text{ind}(\widetilde{\mathbf{u}})\} \rangle.$$

4171 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
4172 `GrB_extract` ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING`
4173 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
4174 result vector, \mathbf{w} , is invalid from this point forward in the sequence.

4175 The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 4176 • If $\text{accum} = \text{GrB_NULL}$, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.
- 4177 • If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$4178 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

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
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.

4245 Return Values

4246	GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-
4247		blocking mode, this indicates that the compatibility tests on
4248		dimensions and domains for the input arguments passed suc-
4249		cessfully. Either way, output matrix C is ready to be used in the
4250		next method of the sequence.
4251	GrB_PANIC	Unknown internal error.
4252	GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the
4253		opaque GraphBLAS objects (input or output) is in an invalid
4254		state caused by a previous execution error. Call <code>GrB_error()</code> to
4255		access any error messages generated by the implementation.
4256	GrB_OUT_OF_MEMORY	Not enough memory available for the operation.
4257	GrB_UNINITIALIZED_OBJECT	One or more of the GraphBLAS objects has not been initialized
4258		by a call to <code>new</code> (or <code>Matrix_dup</code> for matrix parameters).
4259	GrB_INDEX_OUT_OF_BOUNDS	A value in <code>row_indices</code> is greater than or equal to <code>nrows(A)</code> , or
4260		a value in <code>col_indices</code> is greater than or equal to <code>ncols(A)</code> . In
4261		non-blocking mode, this error can be deferred.
4262	GrB_DIMENSION_MISMATCH	Mask and C dimensions are incompatible, <code>nrows</code> \neq <code>nrows(C)</code> , or
4263		<code>ncols</code> \neq <code>ncols(C)</code> .
4264	GrB_DOMAIN_MISMATCH	The domains of the various matrices are incompatible with each
4265		other or the corresponding domains of the accumulation oper-
4266		ator, or the mask's domain is not compatible with <code>bool</code> (in the
4267		case where <code>desc[GrB_MASK].GrB_STRUCTURE</code> is not set).
4268	GrB_NULL_POINTER	Either argument <code>row_indices</code> is a NULL pointer, argument <code>col_indices</code>
4269		is a NULL pointer, or both.

4270 Description

4271 This variant of `GrB_extract` computes the result of extracting a subset of locations from specified
 4272 rows and columns of a GraphBLAS matrix in a specific order: $C = A(\text{row_indices}, \text{col_indices})$; or,
 4273 if an optional binary accumulation operator (\odot) is provided, $C = C \odot A(\text{row_indices}, \text{col_indices})$.
 4274 More explicitly (not accounting for an optional transpose of A):

$$\begin{aligned}
 &C(i, j) = A(\text{row_indices}[i], \text{col_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\
 &C(i, j) = C(i, j) \odot A(\text{row_indices}[i], \text{col_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}
 \end{aligned}$$

4276 Logically, this operation occurs in three steps:

4277 **Setup** The internal matrices and mask used in the computation are formed and their domains
 4278 and dimensions are tested for compatibility.

4279 **Compute** The indicated computations are carried out.

4280 **Output** The result is written into the output matrix, possibly under control of a mask.

4281 Up to three argument matrices are used in the `GrB_extract` operation:

- 4282 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
4283 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)
4284 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4285 The argument matrices and the accumulation operator (if provided) are tested for domain compat-
4286 ibility as follows:

- 4287 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
4288 must be from one of the pre-defined types of Table 3.2.
4289 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.
4290 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4291 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
4292 mulation operator.

4293 Two domains are compatible with each other if values from one domain can be cast to values in
4294 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4295 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4296 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch
4297 error listed above is returned.

4298 From the arguments, the internal matrices, `mask`, and index arrays used in the computation are
4299 formed (\leftarrow denotes copy):

- 4300 1. Matrix $\tilde{C} \leftarrow C$.
4301 2. Two-dimensional mask, \tilde{M} , is computed from argument `Mask` as follows:
4302 (a) If `Mask` = `GrB_NULL`, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
4303 $j < \mathbf{ncols}(C)\} \rangle$.
4304 (b) If `Mask` \neq `GrB_NULL`,
4305 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
4306 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
4307 ii. Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
4308 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
4309 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.
4310 3. Matrix $\tilde{A} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? A^T : A$.

- 4311 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
- 4312 (a) If `row_indices` = `GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$.
- 4313 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{row_indices}[i], \forall i : 0 \leq i < \text{nrows}$.
- 4314 5. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:
- 4315 (a) If `col_indices` = `GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \text{ncols}$.
- 4316 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \text{ncols}$.

4317 The internal matrices and mask are checked for dimension compatibility. The following conditions
4318 must hold:

- 4319 1. $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}(\tilde{\mathbf{M}})$.
- 4320 2. $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}(\tilde{\mathbf{M}})$.
- 4321 3. $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}$.
- 4322 4. $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}$.

4323 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
4324 match error listed above is returned.

4325 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4326 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4327 We are now ready to carry out the extract and any additional associated operations. We describe
4328 this in terms of two intermediate matrices:

- 4329 • $\tilde{\mathbf{T}}$: The matrix holding the extraction from $\tilde{\mathbf{A}}$.
- 4330 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

4331 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

4332
$$\tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \{ (i, j, \tilde{\mathbf{A}}(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j])) \mid \forall (i, j), 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} : (\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j]) \in \text{ind}(\tilde{\mathbf{A}}) \} \rangle.$$

4333 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 $\tilde{\mathbf{J}}$
4334 array is not in the range $[0, \text{ncols}(\tilde{\mathbf{A}}))$, the execution of `GrB_extract` ends and the index out-of-
4335 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred
4336 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from
4337 this point forward in the sequence.

4338 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 4339 • If `accum` = `GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.

4340 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$4341 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4342 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
4343 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4344 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$4345 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$4346 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4349 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4350 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
4351 using what is called a *standard matrix mask and replace*. This is carried out under control of the
4352 mask which acts as a “write mask”.

4353 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
4354 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$4355 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4356 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
4357 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
4358 mask are unchanged:

$$4359 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4360 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
4361 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
4362 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
4363 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
4364 sequence.

4365 4.3.6.3 extract: Column (and row) variant

4366 Extract from one column of a matrix into a vector. Note that with the transpose descriptor for the
4367 source matrix, elements of an arbitrary row of the matrix can be extracted with this function as
4368 well.

4369 C Syntax

```

4370         GrB_Info GrB_extract(GrB_Vector      w,
4371                               const GrB_Vector mask,
4372                               const GrB_BinaryOp accum,
4373                               const GrB_Matrix A,
4374                               const GrB_Index *row_indices,
4375                               GrB_Index nrows,
4376                               GrB_Index col_index,
4377                               const GrB_Descriptor desc);

```

4378 Parameters

4379 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4380 that may be accumulated with the result of the extract operation. On output, this
4381 vector holds the results of the operation.

4382 **mask** (IN) An optional “write” mask that controls which results from this operation are
4383 stored into the output vector **w**. The mask dimensions must match those of the
4384 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4385 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
4386 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
4387 dimensions of **w**), **GrB_NULL** should be specified.

4388 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
4389 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4390 specified.

4391 **A** (IN) The GraphBLAS matrix from which the column subset is extracted.

4392 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations
4393 within the specified column of **A** from which elements are extracted. If elements in
4394 all rows of **A** are to be extracted in order, **GrB_ALL** should be specified. Regardless
4395 of execution mode and return value, this array may be manipulated by the caller
4396 after this operation returns without affecting any deferred computations for this
4397 operation.

4398 **nrows** (IN) The number of indices in the **row_indices** array. Must be equal to **size(w)**.

4399 **col_index** (IN) The index of the column of **A** from which to extract values. It must be in the
4400 range $[0, \mathbf{ncols}(A))$.

4401 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
4402 should be specified. Non-default field/value pairs are listed as follows:
4403

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask .
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector **w** is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

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

GrB_OUT_OF_MEMORY Not enough memory available for operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to **new** (or **dup** for vector or matrix parameters).

GrB_INVALID_INDEX **col_index** is outside the allowable range (i.e., greater than **ncols(A)**).

GrB_INDEX_OUT_OF_BOUNDS A value in **row_indices** is greater than or equal to **nrows(A)**. In non-blocking mode, this error can be deferred.

GrB_DIMENSION_MISMATCH **mask** and **w** dimensions are incompatible, or **nrows** \neq **size(w)**.

GrB_DOMAIN_MISMATCH The domains of the vector or matrix are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with **bool** (in the case where **desc[GrB_MASK].GrB_STRUCTURE** is not set).

GrB_NULL_POINTER Argument **row_indices** is a NULL pointer.

Description

This variant of **GrB_extract** computes the result of extracting a subset of locations (in a specific order) from a specified column of a GraphBLAS matrix: **w** = **A(:, col_index)(row_indices)**; or, if

4431 an optional binary accumulation operator (\odot) is provided, $w = w \odot A(:, \text{col_index})(\text{row_indices})$.
 4432 More explicitly:

$$4433 \quad \begin{aligned} w(i) &= A(\text{row_indices}[i], \text{col_index}) \quad \forall i : 0 \leq i < \text{nrows}, \quad \text{or} \\ w(i) &= w(i) \odot A(\text{row_indices}[i], \text{col_index}) \quad \forall i : 0 \leq i < \text{nrows} \end{aligned}$$

4434 Logically, this operation occurs in three steps:

4435 **Setup** The internal matrices, vectors, and mask used in the computation are formed and their
 4436 domains and dimensions are tested for compatibility.

4437 **Compute** The indicated computations are carried out.

4438 **Output** The result is written into the output vector, possibly under control of a mask.

4439 Up to three argument vectors and matrices are used in this GrB_extract operation:

- 4440 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4441 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4442 3. $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4443 The argument vectors, matrix and the accumulation operator (if provided) are tested for domain
 4444 compatibility as follows:

- 4445 1. If **mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{mask})$
 4446 must be from one of the pre-defined types of Table 3.2.
- 4447 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}(A)$.
- 4448 3. If **accum** is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 4449 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 4450 mulation operator.

4451 Two domains are compatible with each other if values from one domain can be cast to values in
 4452 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 4453 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 4454 any compatibility rule above is violated, execution of GrB_extract ends and the domain mismatch
 4455 error listed above is returned.

4456 From the arguments, the internal vector, matrix, mask, and index array used in the computation
 4457 are formed (\leftarrow denotes copy):

- 4458 1. Vector $\tilde{w} \leftarrow w$.
- 4459 2. One-dimensional mask, \tilde{m} , is computed from argument **mask** as follows:
 4460 (a) If **mask** = GrB_NULL, then $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$.

4461 (b) If $\text{mask} \neq \text{GrB_NULL}$,
 4462 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 4463 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 4464 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
 4465 3. Matrix $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
 4466 4. The internal row index array, $\widetilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
 4467 (a) If `indices = GrB_ALL`, then $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$.
 4468 (b) Otherwise, $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nrows}$.

4469 The internal vector, `mask`, and index array are checked for dimension compatibility. The following
 4470 conditions must hold:

- 4471 1. $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
 4472 2. $\text{size}(\widetilde{\mathbf{w}}) = \text{nrows}$.

4473 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
 4474 match error listed above is returned.

4475 The `col_index` parameter is checked for a valid value. The following condition must hold:

- 4476 1. $0 \leq \text{col_index} < \text{ncols}(\mathbf{A})$

4477 If the rule above is violated, execution of `GrB_extract` ends and the invalid index error listed above
 4478 is returned.

4479 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 4480 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4481 We are now ready to carry out the extract and any additional associated operations. We describe
 4482 this in terms of two intermediate vectors:

- 4483 • $\widetilde{\mathbf{t}}$: The vector holding the extraction from a column of $\widetilde{\mathbf{A}}$.
- 4484 • $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4485 The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$4486 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}, \{(i, \widetilde{\mathbf{A}}(\widetilde{\mathbf{I}}[i], \text{col_index})) \mid \forall i, 0 \leq i < \text{nrows} : (\widetilde{\mathbf{I}}[i], \text{col_index}) \in \text{ind}(\widetilde{\mathbf{A}})\} \rangle.$$

4487 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`
 4488 ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING` mode,
 4489 the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the result
 4490 vector, \mathbf{w} , is invalid from this point forward in the sequence.

4491 The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

4492 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

4493 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$4494 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4495 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4496 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$4497 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

$$4498 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

$$4500 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

4502 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4503 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
4504 using what is called a *standard vector mask and replace*. This is carried out under control of the
4505 mask which acts as a “write mask”.

4506 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
4507 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$4508 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4509 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
4510 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
4511 mask are unchanged:

$$4512 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4513 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
4514 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
4515 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
4516 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
4517 sequence.

4518 4.3.7 assign: Modifying sub-graphs

4519 Assign the contents of a subset of a matrix or vector.

4520 4.3.7.1 assign: Standard vector variant

4521 Assign values from one GraphBLAS vector to a subset of a vector as specified by a set of indices.
4522 The size of the input vector is the same size as the index array provided.

4523 C Syntax

```

4524         GrB_Info GrB_assign(GrB_Vector      w,
4525                             const GrB_Vector mask,
4526                             const GrB_BinaryOp accum,
4527                             const GrB_Vector u,
4528                             const GrB_Index *indices,
4529                             GrB_Index      nindices,
4530                             const GrB_Descriptor desc);

```

4531 Parameters

4532 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4533 that may be accumulated with the result of the assign operation. On output, this
4534 vector holds the results of the operation.

4535 **mask** (IN) An optional “write” mask that controls which results from this operation are
4536 stored into the output vector **w**. The mask dimensions must match those of the
4537 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4538 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
4539 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
4540 dimensions of **w**), **GrB_NULL** should be specified.

4541 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
4542 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4543 specified.

4544 **u** (IN) The GraphBLAS vector whose contents are assigned to a subset of **w**.

4545 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
4546 **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0
4547 to **nindices** – 1, then **GrB_ALL** should be specified. Regardless of execution mode
4548 and return value, this array may be manipulated by the caller after this operation
4549 returns without affecting any deferred computations for this operation. If this
4550 array contains duplicate values, it implies in assignment of more than one value to
4551 the same location which leads to undefined results.

4552 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(u)**.

4553 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
4554 should be specified. Non-default field/value pairs are listed as follows:
4555

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:

$$\begin{aligned}
 w(\text{indices}[i]) &= u(i), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\
 w(\text{indices}[i]) &= w(\text{indices}[i]) \odot u(i), \forall i : 0 \leq i < \text{nindices}.
 \end{aligned}$$

4584 Logically, this operation occurs in three steps:

4585 **Setup** The internal vectors and mask used in the computation are formed and their domains
4586 and dimensions are tested for compatibility.

4587 **Compute** The indicated computations are carried out.

4588 **Output** The result is written into the output vector, possibly under control of a mask.

4589 Up to three argument vectors are used in the `GrB_assign` operation:

- 4590 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 4591 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4592 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4593 The argument vectors and the accumulation operator (if provided) are tested for domain compati-
4594 bility as follows:

- 4595 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
4596 must be from one of the pre-defined types of Table 3.2.
- 4597 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}(\mathbf{u})$.
- 4598 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
4599 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accu-
4600 mulation operator.

4601 Two domains are compatible with each other if values from one domain can be cast to values in
4602 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4603 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4604 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
4605 error listed above is returned.

4606 From the arguments, the internal vectors, mask and index array used in the computation are formed
4607 (\leftarrow denotes copy):

- 4608 1. Vector $\widetilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 4609 2. One-dimensional mask, $\widetilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 4610 (a) If `mask` = `GrB_NULL`, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 4611 (b) If `mask` \neq `GrB_NULL`,
 - 4612 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 4613 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 4614 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.

4615 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

4616 4. The internal index array, $\tilde{\mathbf{I}}$, is computed from argument indices as follows:

4617 (a) If `indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$.

4618 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$.

4619 The internal vector and mask are checked for dimension compatibility. The following conditions
4620 must hold:

4621 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

4622 2. $\text{nindices} = \text{size}(\tilde{\mathbf{u}})$.

4623 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
4624 match error listed above is returned.

4625 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4626 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4627 We are now ready to carry out the assign and any additional associated operations. We describe
4628 this in terms of two intermediate vectors:

- 4629 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{w}}$.
- 4630 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4631 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$4632 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nindices} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4633 At this point, if any value of $\tilde{\mathbf{I}}[i]$ is outside the valid range of indices for vector $\tilde{\mathbf{w}}$, computation
4634 ends and the method returns the index-out-of-bounds error listed above. In `GrB_NONBLOCKING`
4635 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
4636 result vector, \mathbf{w} , is invalid from this point forward in the sequence.

4637 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

- 4638 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$4639 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4640 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
4641 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
4642 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\text{ind}(\tilde{\mathbf{t}})$).

4643 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4644 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$4645 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))),$$

$$4646 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4648 where the difference operator refers to set difference.

- If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$\langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

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.

4.3.7.2 assign: Standard matrix variant

Assign values from one GraphBLAS matrix to a subset of a matrix as specified by a set of indices. The dimensions of the input matrix are the same size as the row and column index arrays provided.

C Syntax

```
GrB_Info GrB_assign(GrB_Matrix      C,
                    const GrB_Matrix Mask,
                    const GrB_BinaryOp accum,
                    const GrB_Matrix A,
```

```

4682         const GrB_Index      *row_indices,
4683         GrB_Index             nrows,
4684         const GrB_Index      *col_indices,
4685         GrB_Index             ncols,
4686         const GrB_Descriptor desc);

```

4687 Parameters

4688 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
4689 that may be accumulated with the result of the assign operation. On output, the
4690 matrix holds the results of the operation.

4691 **Mask** (IN) An optional “write” mask that controls which results from this operation are
4692 stored into the output matrix **C**. The mask dimensions must match those of the
4693 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4694 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
4695 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
4696 dimensions of **C**), **GrB_NULL** should be specified.

4697 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
4698 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4699 specified.

4700 **A** (IN) The GraphBLAS matrix whose contents are assigned to a subset of **C**.

4701 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**
4702 that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** – 1,
4703 then **GrB_ALL** can be specified. Regardless of execution mode and return value,
4704 this array may be manipulated by the caller after this operation returns without
4705 affecting any deferred computations for this operation. If this array contains du-
4706 plicate values, it implies assignment of more than one value to the same location
4707 which leads to undefined results.

4708 **nrows** (IN) The number of values in the **row_indices** array. Must be equal to **nrows(A)**
4709 if **A** is not transposed, or equal to **ncols(A)** if **A** is transposed.

4710 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns
4711 of **C** that are assigned. If all columns of **C** are to be assigned in order from 0
4712 to **ncols** – 1, then **GrB_ALL** should be specified. Regardless of execution mode
4713 and return value, this array may be manipulated by the caller after this operation
4714 returns without affecting any deferred computations for this operation. If this
4715 array contains duplicate values, it implies assignment of more than one value to
4716 the same location which leads to undefined results.

4717 **ncols** (IN) The number of values in **col_indices** array. Must be equal to **ncols(A)** if **A** is
4718 not transposed, or equal to **nrows(A)** if **A** is transposed.

4719
4720
4721

4722

4723

4724
4725
4726
4727
4728

4729

4730
4731
4732
4733

4734

4735
4736

4737
4738
4739

4740
4741

4742
4743
4744
4745

4746
4747

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.

4748 Description

4749 This variant of `GrB_assign` computes the result of assigning the contents of **A** to a subset of rows
 4750 and columns in **C** in a specified order: $\mathbf{C}(\text{row_indices}, \text{col_indices}) = \mathbf{A}$; or, if an optional binary
 4751 accumulation operator (\odot) is provided, $\mathbf{C}(\text{row_indices}, \text{col_indices}) = \mathbf{C}(\text{row_indices}, \text{col_indices}) \odot$
 4752 **A**. More explicitly (not accounting for an optional transpose of **A**):

$$\begin{aligned} & \mathbf{C}(\text{row_indices}[i], \text{col_indices}[j]) = \mathbf{A}(i, j), \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\ 4753 & \mathbf{C}(\text{row_indices}[i], \text{col_indices}[j]) = \mathbf{C}(\text{row_indices}[i], \text{col_indices}[j]) \odot \mathbf{A}(i, j), \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

4754 Logically, this operation occurs in three steps:

4755 **Setup** The internal matrices and mask used in the computation are formed and their domains
 4756 and dimensions are tested for compatibility.

4757 **Compute** The indicated computations are carried out.

4758 **Output** The result is written into the output matrix, possibly under control of a mask.

4759 Up to three argument matrices are used in the `GrB_assign` operation:

- 4760 1. $\mathbf{C} = \langle \mathbf{D}(\mathbf{C}), \text{nrows}(\mathbf{C}), \text{ncols}(\mathbf{C}), \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij})\} \rangle$
- 4761 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)
- 4762 3. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}(\mathbf{A}), \text{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

4763 The argument matrices and the accumulation operator (if provided) are tested for domain compat-
 4764 ibility as follows:

- 4765 1. If **Mask** is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 4766 must be from one of the pre-defined types of Table 3.2.
- 4767 2. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}(\mathbf{A})$.
- 4768 3. If **accum** is not `GrB_NULL`, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 4769 of the accumulation operator and $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 4770 mulation operator.

4771 Two domains are compatible with each other if values from one domain can be cast to values in
 4772 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 4773 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 4774 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
 4775 error listed above is returned.

4776 From the arguments, the internal matrices, mask, and index arrays used in the computation are
 4777 formed (\leftarrow denotes copy):

- 4778 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 4779 2. Two-dimensional mask $\tilde{\mathbf{M}}$ is computed from argument `Mask` as follows:
- 4780 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
4781 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
- 4782 (b) If `Mask \neq GrB_NULL`,
- 4783 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
4784 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
- 4785 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
4786 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
- 4787 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 4788 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 4789 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
- 4790 (a) If `row_indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.
- 4791 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \mathbf{row_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$.
- 4792 5. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:
- 4793 (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.
- 4794 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \mathbf{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

4795 The internal matrices and mask are checked for dimension compatibility. The following conditions
4796 must hold:

- 4797 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 4798 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 4799 3. $\mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}$.
- 4800 4. $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}$.

4801 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
4802 match error listed above is returned.

4803 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4804 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4805 We are now ready to carry out the assign and any additional associated operations. We describe
4806 this in terms of two intermediate vectors:

- 4807 • $\tilde{\mathbf{T}}$: The matrix holding the contents from $\tilde{\mathbf{A}}$ in their destination locations relative to $\tilde{\mathbf{C}}$.
- 4808 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

4809 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$4810 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \tilde{\mathbf{A}}(i, j)) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols} : (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle.$$

4811 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
 4812 $\tilde{\mathbf{J}}$ array is not in the range $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$, the execution of `GrB_assign` ends and the index out-of-
 4813 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred
 4814 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from
 4815 this point forward in the sequence.

4816 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows:

- 4817 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}}$ is defined as

$$4818 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 4819 \quad \{(i, j, Z_{ij}) \mid \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4820 The above expression defines the structure of matrix $\tilde{\mathbf{Z}}$ as follows: We start with the structure
 4821 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
 4822 assigned ($\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$). Finally, we add the structure of $\tilde{\mathbf{T}}$ ($\mathbf{ind}(\tilde{\mathbf{T}})$).

4823 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 4824 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4825 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4826 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}),$$

4827 where the difference operator refers to set difference.

- 4829 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$4830 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4831 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 4832 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4833 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 4834 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4835 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4836 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4837 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 4840 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 4841 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

4875 bool or any of the predefined “built-in” types in Table 3.2. If the default mask
 4876 is desired (i.e., a mask that is all true with the dimensions of a column of C),
 4877 GrB_NULL should be specified.

4878 **accum** (IN) An optional binary operator used for accumulating entries into existing C
 4879 entries. If assignment rather than accumulation is desired, GrB_NULL should be
 4880 specified.

4881 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a column
 4882 of C.

4883 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
 4884 the specified column of C that are to be assigned. If all elements of the column
 4885 in C are to be assigned in order from index 0 to **nrows** – 1, then GrB_ALL should
 4886 be specified. Regardless of execution mode and return value, this array may be
 4887 manipulated by the caller after this operation returns without affecting any de-
 4888 ferred computations for this operation. If this array contains duplicate values, it
 4889 implies in assignment of more than one value to the same location which leads to
 4890 undefined results.

4891 **nrows** (IN) The number of values in **row_indices** array. Must be equal to **size(u)**.

4892 **col_index** (IN) The index of the column in C to assign. Must be in the range [0, **ncols(C)**).

4893 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 4894 should be specified. Non-default field/value pairs are listed as follows:

4895

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 mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.

4897 Return Values

4898 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 4899 blocking mode, this indicates that the compatibility tests on
 4900 dimensions and domains for the input arguments passed suc-
 4901 cessfully. Either way, output matrix C is ready to be used in the
 4902 next method of the sequence.

4903 **GrB_PANIC** Unknown internal error.

4935 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4936 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain
4937 compatibility as follows:

4938 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
4939 must be from one of the pre-defined types of Table 3.2.

4940 2. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}(\mathbf{u})$.

4941 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4942 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
4943 mulation operator.

4944 Two domains are compatible with each other if values from one domain can be cast to values in
4945 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4946 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4947 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
4948 error listed above is returned.

4949 The `col_index` parameter is checked for a valid value. The following condition must hold:

4950 1. $0 \leq \text{col_index} < \mathbf{ncols}(\mathbf{C})$

4951 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above
4952 is returned.

4953 From the arguments, the internal vectors, `mask`, and index array used in the computation are
4954 formed (\leftarrow denotes copy):

4955 1. The vector, $\tilde{\mathbf{c}}$, is extracted from a column of \mathbf{C} as follows:

4956
$$\tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \{(i, C_{ij}) \mid i : 0 \leq i < \mathbf{nrows}(\mathbf{C}), j = \text{col_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

4957 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:

4958 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{nrows}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C})\} \rangle$.

4959 (b) If `mask` \neq `GrB_NULL`,

4960 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,

4961 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.

4962 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

4963 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

4964 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:

4965 (a) If `row_indices` = `GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.

4966 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{row_indices}[i]$, $\forall i : 0 \leq i < \text{nrows}$.

4967 The internal vectors, matrices, and masks are checked for dimension compatibility. The following
4968 conditions must hold:

- 4969 1. $\text{size}(\tilde{\mathbf{c}}) = \text{size}(\tilde{\mathbf{m}})$
- 4970 2. $\text{nrows} = \text{size}(\tilde{\mathbf{u}})$.

4971 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
4972 match error listed above is returned.

4973 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4974 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4975 We are now ready to carry out the assign and any additional associated operations. We describe
4976 this in terms of two intermediate vectors:

- 4977 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{c}}$.
- 4978 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4979 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$4980 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nrows} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4981 At this point, if any value of $\tilde{\mathbf{I}}[i]$ is outside the valid range of indices for vector $\tilde{\mathbf{c}}$, computation
4982 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`
4983 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
4984 result matrix, \mathbf{C} , is invalid from this point forward in the sequence.

4985 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

- 4986 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$4987 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4988 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
4989 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
4990 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\text{ind}(\tilde{\mathbf{t}})$).

4991 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4992 indices in $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{t}}$.

$$4993 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))),$$

$$4994 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4996 where the difference operator refers to set difference.

4997 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$4998 \quad \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4999 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
5000 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$5001 \quad z_i = \tilde{\mathbf{c}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$5002 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5003 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

5006 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

5007 Finally, the set of output values that make up the $\tilde{\mathbf{z}}$ vector are written into the column of the final
5008 result matrix, $\mathbf{C}(:, \text{col_index})$. This is carried out under control of the mask which acts as a “write
5009 mask”.

5010 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in $\mathbf{C}(:, \text{col_index})$ on input to this
5011 operation are deleted and the new contents of the column is given by:

$$5012 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : j \neq \text{col_index}\} \cup \{(i, \text{col_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5013 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
5014 copied into the column of the final result matrix, $\mathbf{C}(:, \text{col_index})$, and elements of this column
5015 that fall outside the set indicated by the mask are unchanged:

$$\begin{aligned} 5016 \quad \mathbf{L}(\mathbf{C}) &= \{(i, j, C_{ij}) : j \neq \text{col_index}\} \cup \\ 5017 &\quad \{(i, \text{col_index}, \tilde{\mathbf{c}}(i)) : i \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \\ 5018 &\quad \{(i, \text{col_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}. \end{aligned}$$

5019 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
5020 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
5021 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may
5022 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

5023 4.3.7.4 assign: Row variant

5024 Assign the contents a vector to a subset of elements in one row of a matrix. Note that since the
5025 output cannot be transposed, a different variant of `assign` is provided to assign to a column of a
5026 matrix.

5027 C Syntax

```
5028         GrB_Info GrB_assign(GrB_Matrix      C,  
5029                             const GrB_Vector mask,  
5030                             const GrB_BinaryOp accum,  
5031                             const GrB_Vector u,  
5032                             GrB_Index      row_index,  
5033                             const GrB_Index *col_indices,  
5034                             GrB_Index      ncols,  
5035                             const GrB_Descriptor desc);
```

5036 Parameters

5037 **C** (INOUT) An existing GraphBLAS Matrix. On input, the matrix provides values
5038 that may be accumulated with the result of the assign operation. On output, this
5039 matrix holds the results of the operation.

5040 **mask** (IN) An optional “write” mask that controls which results from this operation are
5041 stored into the specified row of the output matrix **C**. The mask dimensions must
5042 match those of a single row of the matrix **C**. If the **GrB_STRUCTURE** descriptor
5043 is *not* set for the mask, the domain of the **Mask** matrix must be of type **bool** or
5044 any of the predefined “built-in” types in Table 3.2. If the default mask is desired
5045 (i.e., a mask that is all **true** with the dimensions of a row of **C**), **GrB_NULL** should
5046 be specified.

5047 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
5048 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5049 specified.

5050 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a row of
5051 **C**.

5052 **row_index** (IN) The index of the row in **C** to assign. Must be in the range $[0, \mathbf{nrows}(\mathbf{C})]$.

5053 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
5054 the specified row of **C** that are to be assigned. If all elements of the row in **C** are to
5055 be assigned in order from index 0 to $\mathbf{ncols} - 1$, then **GrB_ALL** should be specified.
5056 Regardless of execution mode and return value, this array may be manipulated by
5057 the caller after this operation returns without affecting any deferred computations
5058 for this operation. If this array contains duplicate values, it implies in assignment
5059 of more than one value to the same location which leads to undefined results.

5060 **ncols** (IN) The number of values in **col_indices** array. Must be equal to **size(u)**.

5061 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
5062 should be specified. Non-default field/value pairs are listed as follows:
5063

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 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 matrix C is ready to be used in the next method of the sequence.
GrB_PANIC	Unknown internal error.
GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.
GrB_OUT_OF_MEMORY	Not enough memory available for operation.
GrB_UNINITIALIZED_OBJECT	One or more of the GraphBLAS objects has not been initialized by a call to new (or dup for vector or matrix parameters).
GrB_INVALID_INDEX	row_index is outside the allowable range (i.e., greater than nrows(C)).
GrB_INDEX_OUT_OF_BOUNDS	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 size and number of columns in C are not the same, or ncols \neq size(u) .
GrB_DOMAIN_MISMATCH	The domains of the matrix and vector are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with bool (in the case where desc[GrB_MASK].GrB_STRUCTURE is not set).
GrB_NULL_POINTER	Argument col_indices 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:

5092 $C(\text{row_index}, :) = u$; or, if an optional binary accumulation operator (\odot) is provided, $C(\text{row_index}, :$
5093 $) = C(\text{row_index}, :) \odot u$. Taking order of `col_indices` into account it is more explicitly written as:

5094 $C(\text{row_index}, \text{col_indices}[j]) = u(j), \forall j : 0 \leq j < \text{ncols}, \text{ or}$
 $C(\text{row_index}, \text{col_indices}[j]) = C(\text{row_index}, \text{col_indices}[j]) \odot u(j), \forall j : 0 \leq j < \text{ncols}$

5095 Logically, this operation occurs in three steps:

5096 **Setup** The internal matrices, vectors and mask used in the computation are formed and their
5097 domains and dimensions are tested for compatibility.

5098 **Compute** The indicated computations are carried out.

5099 **Output** The result is written into the output matrix, possibly under control of a mask.

5100 Up to three argument vectors and matrices are used in this `GrB_assign` operation:

- 5101 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5102 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 5103 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5104 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain
5105 compatibility as follows:

- 5106 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
5107 must be from one of the pre-defined types of Table 3.2.
- 5108 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(u)$.
- 5109 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5110 of the accumulation operator and $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
5111 mulation operator.

5112 Two domains are compatible with each other if values from one domain can be cast to values in
5113 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
5114 compatible with each other. A domain from a user-defined type is only compatible with itself. If
5115 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
5116 error listed above is returned.

5117 The `row_index` parameter is checked for a valid value. The following condition must hold:

- 5118 1. $0 \leq \text{row_index} < \mathbf{nrows}(C)$

5119 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above
5120 is returned.

5121 From the arguments, the internal vectors, mask, and index array used in the computation are
5122 formed (\leftarrow denotes copy):

5123 1. The vector, $\tilde{\mathbf{c}}$, is extracted from a row of \mathbf{C} as follows:

$$5124 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(j, C_{ij}) \mid \forall j : 0 \leq j < \mathbf{ncols}(\mathbf{C}), i = \text{row_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

5125 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:

5126 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{ncols}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{ncols}(\mathbf{C})\} \rangle$.

5127 (b) If `mask \neq GrB_NULL`,

5128 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,

5129 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.

5130 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

5131 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

5132 4. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:

5133 (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.

5134 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

5135 The internal vectors, matrices, and masks are checked for dimension compatibility. The following
5136 conditions must hold:

5137 1. $\mathbf{size}(\tilde{\mathbf{c}}) = \mathbf{size}(\tilde{\mathbf{m}})$

5138 2. $\mathbf{ncols} = \mathbf{size}(\tilde{\mathbf{u}})$.

5139 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
5140 match error listed above is returned.

5141 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
5142 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5143 We are now ready to carry out the assign and any additional associated operations. We describe
5144 this in terms of two intermediate vectors:

- 5145 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{c}}$.
- 5146 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

5147 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$5148 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{J}}[j], \tilde{\mathbf{u}}(j)) \mid \forall j, 0 \leq j < \mathbf{ncols} : j \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle.$$

5149 At this point, if any value of $\tilde{\mathbf{J}}[j]$ is outside the valid range of indices for vector $\tilde{\mathbf{c}}$, computation
5150 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`
5151 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
5152 result matrix, \mathbf{C} , is invalid from this point forward in the sequence.

5153 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

5154 • If $\text{accum} = \text{GrB_NULL}$, then $\tilde{\mathbf{z}}$ is defined as

$$5155 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5156 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
5157 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
5158 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\mathbf{ind}(\tilde{\mathbf{t}})$).

5159 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
5160 indices in $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{t}}$.

$$5161 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5162 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5164 where the difference operator refers to set difference.

5165 • If accum is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$5166 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{c}}), \{(j, z_j) \mid j \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5167 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
5168 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$5169 \quad z_j = \tilde{\mathbf{c}}(j) \odot \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$5170 \quad z_j = \tilde{\mathbf{c}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5171 \quad z_j = \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

5172 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

5173 Finally, the set of output values that make up the $\tilde{\mathbf{z}}$ vector are written into the column of the final
5174 result matrix, $\mathbf{C}(\text{row_index}, :)$. This is carried out under control of the mask which acts as a “write
5175 mask”.

5176 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is set, then any values in $\mathbf{C}(\text{row_index}, :)$ on input to this
5177 operation are deleted and the new contents of the column is given by:

$$5180 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row_index}\} \cup \{(\text{row_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5181 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
5182 copied into the column of the final result matrix, $\mathbf{C}(\text{row_index}, :)$, and elements of this column
5183 that fall outside the set indicated by the mask are unchanged:

$$5184 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row_index}\} \cup$$

$$5185 \quad \{(\text{row_index}, j, \tilde{\mathbf{c}}(j)) : j \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup$$

$$5186 \quad \{(\text{row_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5187 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
5188 of vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
5189 exits with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but may
5190 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

5191 4.3.7.5 assign: Constant vector variant[Scott: NEW CONTENT]

5192 Assign the same value to a specified subset of vector elements. With the use of GrB_ALL, the entire
5193 destination vector can be filled with the constant.

5194 C Syntax

```
5195     GrB_Info GrB_assign(GrB_Vector      w,  
5196                        const GrB_Vector mask,  
5197                        const GrB_BinaryOp accum,  
5198                        <type>          val,  
5199                        const GrB_Index  *indices,  
5200                        GrB_Index        nindices,  
5201                        const GrB_Descriptor desc);
```

```
5202     GrB_Info GrB_assign(GrB_Vector      w,  
5203                        const GrB_Vector mask,  
5204                        const GrB_BinaryOp accum,  
5205                        const GrB_Scalar  s,  
5206                        const GrB_Index  *indices,  
5207                        GrB_Index        nindices,  
5208                        const GrB_Descriptor desc);
```

5209 Parameters

5210 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
5211 that may be accumulated with the result of the assign operation. On output, this
5212 vector holds the results of the operation.

5213 **mask** (IN) An optional “write” mask that controls which results from this operation are
5214 stored into the output vector **w**. The mask dimensions must match those of the
5215 vector **w**. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
5216 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
5217 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
5218 dimensions of **w**), GrB_NULL should be specified.

5219 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
5220 entries. If assignment rather than accumulation is desired, GrB_NULL should be
5221 specified.

5222 **val** (IN) Scalar value to assign to (a subset of) **w**.

5223 **s** (IN) Scalar value to assign to (a subset of) **w**.

5224 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
5225 **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0

5226 to `nindices - 1`, then `GrB_ALL` should be specified. Regardless of execution mode
5227 and return value, this array may be manipulated by the caller after this operation
5228 returns without affecting any deferred computations for this operation. In this
5229 variant, the specific order of the values in the array has no effect on the result.
5230 Unlike other variants, if there are duplicated values in this array the result is still
5231 defined.

5232 **nindices** (IN) The number of values in `indices` array. Must be in the range: `[0, size(w)]`. If
5233 `nindices` is zero, the operation becomes a NO-OP.

5234 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
5235 should be specified. Non-default field/value pairs are listed as follows:

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> .

5238 Return Values

5239 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
5240 blocking mode, this indicates that the compatibility tests on
5241 dimensions and domains for the input arguments passed suc-
5242 cessfully. Either way, output vector `w` is ready to be used in the
5243 next method of the sequence.

5244 **GrB_PANIC** Unknown internal error.

5245 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
5246 opaque GraphBLAS objects (input or output) is in an invalid
5247 state caused by a previous execution error. Call `GrB_error()` to
5248 access any error messages generated by the implementation.

5249 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

5250 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized
5251 by a call to `new` (or `dup` for vector parameters).

5252 **GrB_INDEX_OUT_OF_BOUNDS** A value in `indices` is greater than or equal to `size(w)`. In non-
5253 blocking mode, this can be reported as an execution error.

5254 **GrB_DIMENSION_MISMATCH** `mask` and `w` dimensions are incompatible, or `nindices` is not less
5255 than `size(w)`.

5286 4. If **accum** is not **GrB_NULL**, then either **D(val)** or **D(s)**, depending on the signature of the
 5287 method, must be compatible with **D_{in2}(accum)** of the accumulation operator.

5288 Two domains are compatible with each other if values from one domain can be cast to values in
 5289 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 5290 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 5291 any compatibility rule above is violated, execution of **GrB_assign** ends and the domain mismatch
 5292 error listed above is returned.

5293 From the arguments, the internal vectors, mask and index array used in the computation are formed
 5294 (\leftarrow denotes copy):

- 5295 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5296 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument **mask** as follows:
 - 5297 (a) If **mask** = **GrB_NULL**, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 5298 (b) If **mask** \neq **GrB_NULL**,
 - 5299 i. If **desc[GrB_MASK].GrB_STRUCTURE** is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 5300 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 5301 (c) If **desc[GrB_MASK].GrB_COMP** is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 5302 3. Scalar $\tilde{s} \leftarrow \mathbf{s}$ (**GrB_Scalar** version only).
- 5303 4. The internal index array, $\tilde{\mathbf{I}}$, is computed from argument **indices** as follows:
 - 5304 (a) If **indices** = **GrB_ALL**, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nindices}$.
 - 5305 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \mathbf{indices}[i], \forall i : 0 \leq i < \mathbf{nindices}$.

5306 The internal vector and mask are checked for dimension compatibility. The following conditions
 5307 must hold:

- 5308 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5309 2. $0 \leq \mathbf{nindices} \leq \mathbf{size}(\tilde{\mathbf{w}})$.

5310 If any compatibility rule above is violated, execution of **GrB_assign** ends and the dimension mis-
 5311 match error listed above is returned.

5312 From this point forward, in **GrB_NONBLOCKING** mode, the method can optionally exit with
 5313 **GrB_SUCCESS** return code and defer any computation and/or execution error codes.

5314 We are now ready to carry out the assign and any additional associated operations. We describe
 5315 this in terms of two intermediate vectors:

- 5316 • $\tilde{\mathbf{t}}$: The vector holding the copies of the scalar, either **val** or \tilde{s} , in their destination locations
 5317 relative to $\tilde{\mathbf{w}}$.

5318 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

5319 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows. If a non-opaque scalar \mathbf{val} is provided:

$$5320 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{val}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5321 Correspondingly, if a non-empty `GrB_Scalar` \tilde{s} is provided (i.e., $\mathbf{size}(\tilde{s}) = 1$):

$$5322 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}(\tilde{s})) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5323 Finally, if an empty `GrB_Scalar` \tilde{s} is provided (i.e., $\mathbf{size}(\tilde{s}) = 0$):

$$5324 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \emptyset \rangle.$$

5325 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
 5326 is not in the range $[0, \mathbf{size}(\tilde{\mathbf{w}}))$, the execution of `GrB_assign` ends and the index out-of-bounds
 5327 error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a
 5328 sequence-terminating `GrB_wait()` is called. Regardless, the result vector, \mathbf{w} , is invalid from this
 5329 point forward in the sequence.

5330 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

5331 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$5332 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5333 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
 5334 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
 5335 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\mathbf{ind}(\tilde{\mathbf{t}})$).

5336 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 5337 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$5338 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5340 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5341 where the difference operator refers to set difference. We note that in this case of assigning
 5342 a constant, $\{\tilde{\mathbf{I}}[k], \forall k\}$ and $\mathbf{ind}(\tilde{\mathbf{t}})$ are identical.

5343 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$5344 \quad \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5345 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 5346 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$5347 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$5349 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5351 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

5352 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

5353 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 5354 using what is called a *standard vector mask and replace*. This is carried out under control of the
 5355 mask which acts as a “write mask”.

- 5356 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{w} on input to this operation are
 5357 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$5358 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 5359 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 5360 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 5361 mask are unchanged:

$$5362 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5363 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 5364 of vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 5365 exits with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but
 5366 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 5367 sequence.

5368 4.3.7.6 assign: Constant matrix variant[Scott: NEW CONTENT]

5369 Assign the same value to a specified subset of matrix elements. With the use of GrB_ALL, the
 5370 entire destination matrix can be filled with the constant.

5371 C Syntax

```
5372      GrB_Info GrB_assign(GrB_Matrix      C,
5373                          const GrB_Matrix Mask,
5374                          const GrB_BinaryOp accum,
5375                          <type>         val,
5376                          const GrB_Index *row_indices,
5377                          GrB_Index      nrows,
5378                          const GrB_Index *col_indices,
5379                          GrB_Index      ncols,
5380                          const GrB_Descriptor desc);
```

```
5381      GrB_Info GrB_assign(GrB_Matrix      C,
5382                          const GrB_Matrix Mask,
5383                          const GrB_BinaryOp accum,
5384                          const GrB_Scalar s,
5385                          const GrB_Index *row_indices,
5386                          GrB_Index      nrows,
```

```

5387         const GrB_Index      *col_indices,
5388         GrB_Index             ncols,
5389         const GrB_Descriptor  desc);

```

5390 Parameters

5391 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
5392 that may be accumulated with the result of the assign operation. On output, the
5393 matrix holds the results of the operation.

5394 **Mask** (IN) An optional “write” mask that controls which results from this operation are
5395 stored into the output matrix **C**. The mask dimensions must match those of the
5396 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
5397 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
5398 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
5399 dimensions of **C**), **GrB_NULL** should be specified.

5400 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
5401 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5402 specified.

5403 **val** (IN) Scalar value to assign to (a subset of) **C**.

5404 **s** (IN) Scalar value to assign to (a subset of) **C**.

5405 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**
5406 that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** − 1,
5407 then **GrB_ALL** can be specified. Regardless of execution mode and return value,
5408 this array may be manipulated by the caller after this operation returns without
5409 affecting any deferred computations for this operation. Unlike other variants, if
5410 there are duplicated values in this array the result is still defined.

5411 **nrows** (IN) The number of values in **row_indices** array. Must be in the range: [0, **nrows**(**C**)].
5412 If **nrows** is zero, the operation becomes a NO-OP.

5413 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns of **C**
5414 that are assigned. If all columns of **C** are to be assigned in order from 0 to **ncols** − 1,
5415 then **GrB_ALL** should be specified. Regardless of execution mode and return value,
5416 this array may be manipulated by the caller after this operation returns without
5417 affecting any deferred computations for this operation. Unlike other variants, if
5418 there are duplicated values in this array the result is still defined.

5419 **ncols** (IN) The number of values in **col_indices** array. Must be in the range: [0, **ncols**(**C**)].
5420 If **ncols** is zero, the operation becomes a NO-OP.

5421 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
5422 should be specified. Non-default field/value pairs are listed as follows:

5423

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`

5453 or $C(\text{row_indices}, \text{col_indices}) = s$ is performed. If an optional binary accumulation operator
 5454 (\odot) is provided, then either $C(\text{row_indices}, \text{col_indices}) = C(\text{row_indices}, \text{col_indices}) \odot \text{val}$ or
 5455 $C(\text{row_indices}, \text{col_indices}) = C(\text{row_indices}, \text{col_indices}) \odot s$ is performed. More explicitly, if a
 5456 non-opaque value val is provided:

$$\begin{aligned} & C(\text{row_indices}[i], \text{col_indices}[j]) = \text{val}, \text{ or} \\ 5457 & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot \text{val} \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5458 Correspondingly, if a `GrB_Scalar` s is provided:

$$\begin{aligned} & C(\text{row_indices}[i], \text{col_indices}[j]) = s, \text{ or} \\ 5459 & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot s \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5460 Logically, this operation occurs in three steps:

5461 Setup The internal vectors and mask used in the computation are formed and their domains
 5462 and dimensions are tested for compatibility.

5463 Compute The indicated computations are carried out.

5464 Output The result is written into the output matrix, possibly under control of a mask.

5465 Up to two argument matrices are used in the `GrB_assign` operation:

- 5466 1. $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5467 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)

5468 The argument scalar, matrices, and the accumulation operator (if provided) are tested for domain
 5469 compatibility as follows:

- 5470 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 5471 must be from one of the pre-defined types of Table 3.2.
- 5472 2. $\mathbf{D}(C)$ must be compatible with either $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$, depending on the signature of the
 5473 method.
- 5474 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 5475 of the accumulation operator.
- 5476 4. If `accum` is not `GrB_NULL`, then either $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$, depending on the signature of the
 5477 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, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\text{bool})\mathbf{Mask}(i, j) = \text{true}\} \rangle$.
 - (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, $\tilde{\mathbf{I}}[i] = \text{row_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$.
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, $\tilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

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.

5510 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 5511 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5512 We are now ready to carry out the assign and any additional associated operations. We describe
 5513 this in terms of two intermediate matrices:

- 5514 • $\tilde{\mathbf{T}}$: The matrix holding the copies of the scalar, either `val` or \tilde{s} , in their destination locations
 5515 relative to $\tilde{\mathbf{C}}$.
- 5516 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

5517 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows. If a non-opaque scalar `val` is provided:

$$\begin{aligned} 5518 \quad \tilde{\mathbf{T}} = & \langle \mathbf{D}(\text{val}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ & \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle. \end{aligned}$$

5519 Correspondingly, if a non-empty `GrB_Scalar` \tilde{s} is provided (i.e., `size`(\tilde{s}) = 1):

$$\begin{aligned} 5520 \quad \tilde{\mathbf{T}} = & \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ & \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}(\tilde{s})) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle. \end{aligned}$$

5521 Finally, if an empty `GrB_Scalar` \tilde{s} is provided (i.e., `size`(\tilde{s}) = 0):

$$5522 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \emptyset \rangle.$$

5523 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
 5524 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
 5525 $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$, the execution of `GrB_assign` ends and the index out-of-bounds error listed above is
 5526 generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating
 5527 `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from this point forward in the
 5528 sequence.

5529 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows:

- 5530 • If `accum` = `GrB_NULL`, then $\tilde{\mathbf{Z}}$ is defined as

$$\begin{aligned} 5531 \quad \tilde{\mathbf{Z}} = & \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 5532 \quad & \{(i, j, Z_{ij}) \mid \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}}))\} \rangle. \end{aligned}$$

5533 The above expression defines the structure of matrix $\tilde{\mathbf{Z}}$ as follows: We start with the structure
 5534 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
 5535 assigned ($\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$). Finally, we add the structure of $\tilde{\mathbf{T}}$ ($\mathbf{ind}(\tilde{\mathbf{T}})$).

5536 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 5537 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 5538 \quad Z_{ij} = & \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5539 \quad & \\ 5540 \quad & \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}), \end{aligned}$$

5541 where the difference operator refers to set difference. We note that, in this particular case of
 5542 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.

5543 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$5544 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5545 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
5546 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$5547 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$5548 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$5550 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5552 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

5553 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
5554 using what is called a *standard matrix mask and replace*. This is carried out under control of the
5555 mask which acts as a “write mask”.

5556 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
5557 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$5558 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5559 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
5560 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
5561 mask are unchanged:

$$5562 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5563 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
5564 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
5565 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
5566 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
5567 sequence.

5568 4.3.8 `apply`: Apply a function to the elements of an object

5569 Computes the transformation of the values of the elements of a vector or a matrix using a unary
5570 function, or a binary function where one argument is bound to a scalar.

5571 4.3.8.1 `apply`: Vector variant

5572 Computes the transformation of the values of the elements of a vector using a unary function.

5573 C Syntax

```

5574      GrB_Info GrB_apply(GrB_Vector      w,
5575                          const GrB_Vector mask,
5576                          const GrB_BinaryOp accum,
5577                          const GrB_UnaryOp op,
5578                          const GrB_Vector u,
5579                          const GrB_Descriptor desc);

```

5580 Parameters

5581 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
5582 that may be accumulated with the result of the apply operation. On output, this
5583 vector holds the results of the operation.

5584 **mask** (IN) An optional “write” mask that controls which results from this operation are
5585 stored into the output vector **w**. The mask dimensions must match those of the
5586 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
5587 of the mask vector must be of type **bool** or any of the predefined “built-in” types
5588 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
5589 dimensions of **w**), **GrB_NULL** should be specified.

5590 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
5591 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5592 specified.

5593 **op** (IN) A unary operator applied to each element of input vector **u**.

5594 **u** (IN) The GraphBLAS vector to which the unary function is applied.

5595 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
5596 should be specified. Non-default field/value pairs are listed as follows:

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 .

5599 Return Values

5600 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
5601 blocking mode, this indicates that the compatibility tests on di-
5602 mensions and domains for the input arguments passed successfully.

5603 Either way, output vector w is ready to be used in the next method
 5604 of the sequence.

5605 **GrB_PANIC** Unknown internal error.

5606 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 5607 GraphBLAS objects (input or output) is in an invalid state caused
 5608 by a previous execution error. Call **GrB_error()** to access any error
 5609 messages generated by the implementation.

5610 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

5611 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 5612 a call to **new** (or **dup** for vector parameters).

5613 **GrB_DIMENSION_MISMATCH** $mask$, w and/or u dimensions are incompatible.

5614 **GrB_DOMAIN_MISMATCH** The domains of the various vectors are incompatible with the corre-
 5615 sponding domains of the accumulation operator or unary function,
 5616 or the mask's domain is not compatible with **bool** (in the case where
 5617 $desc[GrB_MASK].GrB_STRUCTURE$ is not set).

5618 **Description**

5619 This variant of **GrB_apply** computes the result of applying a unary function to the elements of a
 5620 GraphBLAS vector: $w = f(u)$; or, if an optional binary accumulation operator (\odot) is provided,
 5621 $w = w \odot f(u)$.

5622 Logically, this operation occurs in three steps:

5623 **Setup** The internal vectors and mask used in the computation are formed and their domains
 5624 and dimensions are tested for compatibility.

5625 **Compute** The indicated computations are carried out.

5626 **Output** The result is written into the output vector, possibly under control of a mask.

5627 Up to three argument vectors are used in this **GrB_apply** operation:

- 5628 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 5629 2. $mask = \langle \mathbf{D}(mask), \mathbf{size}(mask), \mathbf{L}(mask) = \{(i, m_i)\} \rangle$ (optional)
- 5630 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5631 The argument vectors, unary operator and the accumulation operator (if provided) are tested for
 5632 domain compatibility as follows:

- 5633 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
5634 must be from one of the pre-defined types of Table 3.2.
- 5635 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the unary operator.
- 5636 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5637 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the unary operator must be compatible with
5638 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 5639 4. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in}(\text{op})$.

5640 Two domains are compatible with each other if values from one domain can be cast to values in
5641 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
5642 compatible with each other. A domain from a user-defined type is only compatible with itself. If
5643 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
5644 error listed above is returned.

5645 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
5646 denotes copy):

- 5647 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5648 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 5649 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
 - 5650 (b) If `mask` \neq `GrB_NULL`,
 - 5651 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 - 5652 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 5653 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 5654 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

5655 The internal vectors and masks are checked for dimension compatibility. The following conditions
5656 must hold:

- 5657 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$
- 5658 2. $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$.

5659 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
5660 error listed above is returned.

5661 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
5662 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5663 We are now ready to carry out the apply and any additional associated operations. We describe
5664 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.

5697 4.3.8.2 apply: Matrix variant

5698 Computes the transformation of the values of the elements of a matrix using a unary function.

5699 C Syntax

```
5700      GrB_Info GrB_apply(GrB_Matrix      C,
5701                        const GrB_Matrix  Mask,
5702                        const GrB_BinaryOp accum,
5703                        const GrB_UnaryOp  op,
5704                        const GrB_Matrix  A,
5705                        const GrB_Descriptor desc);
```

5706 Parameters

5707 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
5708 that may be accumulated with the result of the apply operation. On output, the
5709 matrix holds the results of the operation.

5710 **Mask** (IN) An optional “write” mask that controls which results from this operation are
5711 stored into the output matrix C. The mask dimensions must match those of the
5712 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
5713 of the Mask matrix must be of type bool or any of the predefined “built-in” types
5714 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
5715 dimensions of C), GrB_NULL should be specified.

5716 **accum** (IN) An optional binary operator used for accumulating entries into existing C
5717 entries. If assignment rather than accumulation is desired, GrB_NULL should be
5718 specified.

5719 **op** (IN) A unary operator applied to each element of input matrix A.

5720 **A** (IN) The GraphBLAS matrix to which the unary function is applied.

5721 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
5722 should be specified. Non-default field/value pairs are listed as follows:

5723

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.

5724

5725 Return Values

5726	GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-
5727		blocking mode, this indicates that the compatibility tests on
5728		dimensions and domains for the input arguments passed suc-
5729		cessfully. Either way, output matrix C is ready to be used in the
5730		next method of the sequence.
5731	GrB_PANIC	Unknown internal error.
5732	GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the
5733		opaque GraphBLAS objects (input or output) is in an invalid
5734		state caused by a previous execution error. Call GrB_error() to
5735		access any error messages generated by the implementation.
5736	GrB_OUT_OF_MEMORY	Not enough memory available for the operation.
5737	GrB_UNINITIALIZED_OBJECT	One or more of the GraphBLAS objects has not been initialized
5738		by a call to new (or Matrix_dup for matrix parameters).
5739	GrB_DIMENSION_MISMATCH	Mask and C dimensions are incompatible, nrows \neq nrows (C), or
5740		ncols \neq ncols (C).
5741	GrB_DOMAIN_MISMATCH	The domains of the various matrices are incompatible with the
5742		corresponding domains of the accumulation operator or unary
5743		function, or the mask's domain is not compatible with bool (in
5744		the case where desc [GrB_MASK]. GrB_STRUCTURE is not set).

5745 Description

5746 This variant of **GrB_apply** computes the result of applying a unary function to the elements of a
 5747 GraphBLAS matrix: $C = f(A)$; or, if an optional binary accumulation operator (\odot) is provided,
 5748 $C = C \odot f(A)$.

5749 Logically, this operation occurs in three steps:

5750 **Setup** The internal matrices and mask used in the computation are formed and their domains
 5751 and dimensions are tested for compatibility.

5752 **Compute** The indicated computations are carried out.

5753 **Output** The result is written into the output matrix, possibly under control of a mask.

5754 Up to three argument matrices are used in the **GrB_apply** operation:

- 5755 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5756 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)

5757 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

5758 The argument matrices, unary operator and the accumulation operator (if provided) are tested for
5759 domain compatibility as follows:

- 5760 1. If **Mask** is not **GrB_NULL**, and **desc[GrB_MASK].GrB_STRUCTURE** is not set, then $\mathbf{D}(\text{Mask})$
5761 must be from one of the pre-defined types of Table 3.2.
- 5762 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the unary operator.
- 5763 3. If **accum** is not **GrB_NULL**, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5764 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the unary operator must be compatible with
5765 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 5766 4. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in}(\text{op})$ of the unary operator.

5767 Two domains are compatible with each other if values from one domain can be cast to values in
5768 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
5769 compatible with each other. A domain from a user-defined type is only compatible with itself. If
5770 any compatibility rule above is violated, execution of **GrB_apply** ends and the domain mismatch
5771 error listed above is returned.

5772 From the argument matrices, the internal matrices, mask, and index arrays used in the computation
5773 are formed (\leftarrow denotes copy):

- 5774 1. Matrix $\tilde{C} \leftarrow C$.
- 5775 2. Two-dimensional mask, \tilde{M} , is computed from argument **Mask** as follows:
 - 5776 (a) If **Mask** = **GrB_NULL**, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
5777 $j < \mathbf{ncols}(C)\} \rangle$.
 - 5778 (b) If **Mask** \neq **GrB_NULL**,
 - 5779 i. If **desc[GrB_MASK].GrB_STRUCTURE** is set, then $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
5780 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
 - 5781 ii. Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
5782 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
 - 5783 (c) If **desc[GrB_MASK].GrB_COMP** is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.
- 5784 3. Matrix $\tilde{A} \leftarrow \text{desc[GrB_INP0].GrB_TRAN} ? A^T : A$.

5785 The internal matrices and mask are checked for dimension compatibility. The following conditions
5786 must hold:

- 5787 1. $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{M})$.
- 5788 2. $\mathbf{ncols}(\tilde{C}) = \mathbf{ncols}(\tilde{M})$.
- 5789 3. $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{A})$.

5790 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

5791 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
5792 error listed above is returned.

5793 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
5794 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5795 We are now ready to carry out the apply and any additional associated operations. We describe
5796 this in terms of two intermediate matrices:

- 5797 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the unary operator to the input matrix $\tilde{\mathbf{A}}$.
- 5798 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

5799 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$5800 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

5801 where $f = \mathbf{f}(\text{op})$.

5802 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 5803 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 5804 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$5805 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5806 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
5807 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$5808 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$5809 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$5811 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5812 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

5814 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
5815 using what is called a *standard matrix mask and replace*. This is carried out under control of the
5816 mask which acts as a “write mask”.

- 5817 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
5818 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$5819 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 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,
```



```

5856             const GrB_BinaryOp      accum,
5857             const GrB_BinaryOp      op,
5858             const GrB_Vector        u,
5859             <type>                  val,
5860             const GrB_Descriptor    desc);

5861 // bind-second + GraphBLAS scalar
5862 GrB_Info GrB_apply(GrB_Vector        w,
5863                   const GrB_Vector    mask,
5864                   const GrB_BinaryOp  accum,
5865                   const GrB_BinaryOp  op,
5866                   const GrB_Vector    u,
5867                   const GrB_Scalar    s,
5868                   const GrB_Descriptor desc);

```

5869 Parameters

5870 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
5871 that may be accumulated with the result of the apply operation. On output, this
5872 vector holds the results of the operation.

5873 **mask** (IN) An optional “write” mask that controls which results from this operation are
5874 stored into the output vector **w**. The mask dimensions must match those of the
5875 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
5876 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
5877 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
5878 dimensions of **w**), **GrB_NULL** should be specified.

5879 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
5880 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5881 specified.

5882 **op** (IN) A binary operator applied to each element of input vector, **u**, and the scalar
5883 value, **val**.

5884 **u** (IN) The GraphBLAS vector whose elements are passed to the binary operator as
5885 the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)
5886 argument in the *bind-second* variant.

5887 **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)
5888 argument in the *bind-first* variant, or the right-hand (second) argument in the
5889 *bind-second* variant.

5890 **s** (IN) A GraphBLAS scalar that is passed to the binary operator as the left-hand
5891 (first) argument in the *bind-first* variant, or the right-hand (second) argument in
5892 the *bind-second* variant. It must not be empty.

5893 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
5894 should be specified. Non-default field/value pairs are listed as follows:

5895

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 .

5896

5897 Return Values

5898 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
5899 blocking mode, this indicates that the compatibility tests on di-
5900 mensions and domains for the input arguments passed successfully.
5901 Either way, output vector **w** is ready to be used in the next method
5902 of the sequence.

5903 GrB_PANIC Unknown internal error.

5904 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
5905 GraphBLAS objects (input or output) is in an invalid state caused
5906 by a previous execution error. Call GrB_error() to access any error
5907 messages generated by the implementation.

5908 GrB_OUT_OF_MEMORY Not enough memory available for operation.

5909 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
5910 a call to new (or dup for vector parameters).

5911 GrB_DIMENSION_MISMATCH **mask**, **w** and/or **u** dimensions are incompatible.

5912 GrB_DOMAIN_MISMATCH The domains of the various vectors and scalar are incompatible with
5913 the corresponding domains of the binary operator or accumulation
5914 operator, or the **mask**'s domain is not compatible with **bool** (in the
5915 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

5916 GrB_EMPTY_OBJECT The GrB_Scalar **s** used in the call is empty (**nvals(s) = 0**) and
5917 therefore a value cannot be passed to the binary operator.

5918 Description

5919 This variant of GrB_apply computes the result of applying a binary operator to the elements of a
5920 GraphBLAS vector each composed with a scalar constant, either **val** or **s**:

5921 bind-first: $w = f(\text{val}, u)$ or $w = f(s, u)$

5922 bind-second: $w = f(u, \text{val})$ or $w = f(u, s)$,

5923 or if an optional binary accumulation operator (\odot) is provided:

5924 bind-first: $w = w \odot f(\text{val}, u)$ or $w = w \odot f(s, u)$

5925 bind-second: $w = w \odot f(u, \text{val})$ or $w = w \odot f(u, s)$.

5926 Logically, this operation occurs in three steps:

5927 **Setup** The internal vectors and mask used in the computation are formed and their domains
5928 and dimensions are tested for compatibility.

5929 **Compute** The indicated computations are carried out.

5930 **Output** The result is written into the output vector, possibly under control of a mask.

5931 Up to three argument vectors are used in this GrB_apply operation:

5932 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$

5933 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)

5934 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5935 The argument scalar, vectors, binary operator and the accumulation operator (if provided) are
5936 tested for domain compatibility as follows:

5937 1. If **mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{mask})$
5938 must be from one of the pre-defined types of Table 3.2.

5939 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the binary operator.

5940 3. If **accum** is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5941 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the binary operator must be compatible with
5942 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

5943 4. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.

5944 5. If bind-first:

5945 (a) $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the binary operator.

5946 (b) If the non-opaque scalar **val** is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$
5947 of the binary operator.

5948 (c) If the GrB_Scalar **s** is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the
5949 binary operator.

- 5950 6. If bind-second:
- 5951 (a) $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the binary operator.
- 5952 (b) If the non-opaque scalar \mathbf{val} is provided, then $\mathbf{D}(\mathbf{val})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$
- 5953 of the binary operator.
- 5954 (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
- 5955 binary operator.

5956 Two domains are compatible with each other if values from one domain can be cast to values in

5957 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all

5958 compatible with each other. A domain from a user-defined type is only compatible with itself. If

5959 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch

5960 error listed above is returned.

5961 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow

5962 denotes copy):

- 5963 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5964 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
- 5965 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
- 5966 (b) If `mask \neq GrB_NULL`,
- 5967 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
- 5968 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
- 5969 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 5970 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 5971 4. Scalar $\tilde{\mathbf{s}} \leftarrow \mathbf{s}$ (GraphBLAS scalar case).

5972 The internal vectors and masks are checked for dimension compatibility. The following conditions

5973 must hold:

- 5974 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5975 2. $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{w}})$.

5976 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch

5977 error listed above is returned.

5978 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with

5979 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5980 If an empty `GrB_Scalar` $\tilde{\mathbf{s}}$ is provided ($\mathbf{nvals}(\tilde{\mathbf{s}}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.

5981 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

5982 `val` with the same domain as $\tilde{\mathbf{s}}$ and set `val = val($\tilde{\mathbf{s}}$)`.

5983 We are now ready to carry out the apply and any additional associated operations. We describe

5984 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.

6018 4.3.8.4 apply: Matrix-BinaryOp variants[Scott: NEW CONTENT]

6019 Computes the transformation of the values of the stored elements of a matrix using a binary
6020 operator and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the
6021 first argument to the binary operator and stored elements of the matrix are passed as the second
6022 argument. In the *bind-second* variant, the elements of the matrix are passed as the first argument
6023 and the specified scalar value is passed as the second argument. The scalar can be passed either as
6024 a non-opaque variable or as a GrB_Scalar object.

6025 C Syntax

```
6026 // bind-first + scalar value
6027 GrB_Info GrB_apply(GrB_Matrix      C,
6028                   const GrB_Matrix Mask,
6029                   const GrB_BinaryOp accum,
6030                   const GrB_BinaryOp op,
6031                   <type>           val,
6032                   const GrB_Matrix A,
6033                   const GrB_Descriptor desc);
```

```
6034 // bind-first + GraphBLAS scalar
6035 GrB_Info GrB_apply(GrB_Matrix      C,
6036                   const GrB_Matrix Mask,
6037                   const GrB_BinaryOp accum,
6038                   const GrB_BinaryOp op,
6039                   const GrB_Scalar s,
6040                   const GrB_Matrix A,
6041                   const GrB_Descriptor desc);
```

```
6042 // bind-second + scalar value
6043 GrB_Info GrB_apply(GrB_Matrix      C,
6044                   const GrB_Matrix Mask,
6045                   const GrB_BinaryOp accum,
6046                   const GrB_BinaryOp op,
6047                   const GrB_Matrix A,
6048                   <type>           val,
6049                   const GrB_Descriptor desc);
```

```
6050 // bind-second + GraphBLAS scalar
6051 GrB_Info GrB_apply(GrB_Matrix      C,
6052                   const GrB_Matrix Mask,
6053                   const GrB_BinaryOp accum,
6054                   const GrB_BinaryOp op,
6055                   const GrB_Matrix A,
```

```

6056         const GrB_Scalar      s,
6057         const GrB_Descriptor desc);

```

6058 Parameters

6059 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
6060 that may be accumulated with the result of the apply operation. On output, the
6061 matrix holds the results of the operation.

6062 **Mask** (IN) An optional “write” mask that controls which results from this operation are
6063 stored into the output matrix C. The mask dimensions must match those of the
6064 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
6065 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types
6066 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
6067 dimensions of C), `GrB_NULL` should be specified.

6068 **accum** (IN) An optional binary operator used for accumulating entries into existing C
6069 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
6070 specified.

6071 **op** (IN) A binary operator applied to each element of input matrix, A, with the element
6072 of the input matrix used as the left-hand argument, and the scalar value, `val`, used
6073 as the right-hand argument.

6074 **A** (IN) The GraphBLAS matrix whose elements are passed to the binary operator as
6075 the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)
6076 argument in the *bind-second* variant.

6077 **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)
6078 argument in the *bind-first* variant, or the right-hand (second) argument in the
6079 *bind-second* variant.

6080 **s** (IN) GraphBLAS scalar value that is passed to the binary operator as the left-hand
6081 (first) argument in the *bind-first* variant, or the right-hand (second) argument in
6082 the *bind-second* variant. It must not be empty.

6083 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
6084 should be specified. Non-default field/value pairs are listed as follows:
6085

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.

6113 Description

6114 This variant of `GrB_apply` computes the result of applying a binary operator to the elements of a
 6115 GraphBLAS matrix each composed with a scalar constant, `val` or `s`:

6116 bind-first: $C = f(\text{val}, A)$ or $C = f(s, A)$

6117 bind-second: $C = f(A, \text{val})$ or $C = f(A, s)$,

6118 or if an optional binary accumulation operator (\odot) is provided:

6119 bind-first: $C = C \odot f(\text{val}, A)$ or $C = C \odot f(s, A)$

6120 bind-second: $C = C \odot f(A, \text{val})$ or $C = C \odot f(A, s)$.

6121 Logically, this operation occurs in three steps:

6122 **Setup** The internal matrices and mask used in the computation are formed and their domains
 6123 and dimensions are tested for compatibility.

6124 **Compute** The indicated computations are carried out.

6125 **Output** The result is written into the output matrix, possibly under control of a mask.

6126 Up to three argument matrices are used in the `GrB_apply` operation:

- 6127 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6128 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)
- 6129 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6130 The argument scalar, matrices, binary operator and the accumulation operator (if provided) are
 6131 tested for domain compatibility as follows:

- 6132 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 6133 must be from one of the pre-defined types of Table 3.2.
- 6134 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the binary operator.
- 6135 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 6136 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the binary operator must be compatible with
 6137 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 6138 4. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.
- 6139 5. If bind-first:
 6140 (a) $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the binary operator.

6141 (b) If the non-opaque scalar val is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$
 6142 of the binary operator.

6143 (c) If the `GrB_Scalar` s is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the
 6144 binary operator.

6145 6. If `bind-second`:

6146 (a) $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.

6147 (b) If the non-opaque scalar val is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$
 6148 of the binary operator.

6149 (c) If the `GrB_Scalar` s is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the
 6150 binary operator.

6151 Two domains are compatible with each other if values from one domain can be cast to values in
 6152 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 6153 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 6154 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
 6155 error listed above is returned.

6156 From the argument matrices, the internal matrices, mask, and index arrays used in the computation
 6157 are formed (\leftarrow denotes copy):

6158 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.

6159 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:

6160 (a) If `Mask` = `GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 6161 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.

6162 (b) If `Mask` \neq `GrB_NULL`,

6163 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
 6164 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,

6165 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
 6166 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.

6167 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.

6168 3. Matrix $\tilde{\mathbf{A}}$ is computed from argument `A` as follows:

6169 `bind-first`: $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP1}].\text{GrB_TRAN} ? A^T : A$

6170 `bind-second`: $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? A^T : A$

6171 4. Scalar $\tilde{s} \leftarrow s$ (`GraphBLAS` scalar case).

6172 The internal matrices and mask are checked for dimension compatibility. The following conditions
 6173 must hold:

6174 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.

6175 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.

6176 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.

6177 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

6178 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
6179 error listed above is returned.

6180 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6181 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6182 If an empty `GrB_Scalar` \tilde{s} is provided ($\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6183 If a non-empty `GrB_Scalar`, \tilde{s} , is provided (i.e., $\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable
6184 \mathbf{val} with the same domain as \tilde{s} and set $\mathbf{val} = \mathbf{val}(\tilde{s})$.

6185 We are now ready to carry out the apply and any additional associated operations. We describe
6186 this in terms of two intermediate matrices:

- 6187 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the binary operator to the input matrix $\tilde{\mathbf{A}}$.
- 6188 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

6189 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as one of the following:

6190 bind-first: $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\mathbf{val}, \tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

6191 bind-second: $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j), \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

6192 where $f = \mathbf{f}(\mathbf{op})$.

6193 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 6194 • If $\mathbf{accum} = \mathbf{GrB_NULL}$, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 6195 • If \mathbf{accum} is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$6196 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6197 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
6198 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$6199 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$6200 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$6201 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

6202 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

6205 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 6206 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 6207 mask which acts as a “write mask”.

- 6208 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
 6209 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$6210 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6211 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 6212 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 6213 mask are unchanged:

$$6214 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6215 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 6216 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 6217 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
 6218 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 6219 sequence.

6220 4.3.8.5 apply: Vector index unary operator variant[Scott: NEW CONTENT]

6221 Computes the transformation of the values of the stored elements of a vector using an index unary
 6222 operator that is a function of the stored value, its location indices, and an user provided scalar
 6223 value. The scalar can be passed either as a non-opaque variable or as a GrB_Scalar object.

6224 C Syntax

```
6225      GrB_Info GrB_apply(GrB_Vector          w,
6226                        const GrB_Vector      mask,
6227                        const GrB_BinaryOp     accum,
6228                        const GrB_IndexUnaryOp op,
6229                        const GrB_Vector      u,
6230                        <type>                 val,
6231                        const GrB_Descriptor   desc);
```

```
6232      GrB_Info GrB_apply(GrB_Vector          w,
6233                        const GrB_Vector      mask,
6234                        const GrB_BinaryOp     accum,
6235                        const GrB_IndexUnaryOp op,
6236                        const GrB_Vector      u,
6237                        const GrB_Scalar      s,
6238                        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
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.

6270 GrB_PANIC Unknown internal error.

6271 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
6272 opaque GraphBLAS objects (input or output) is in an invalid
6273 state caused by a previous execution error. Call GrB_error() to
6274 access any error messages generated by the implementation.

6275 GrB_OUT_OF_MEMORY Not enough memory available for operation.

6276 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
6277 by a call to new (or another constructor).

6278 GrB_DIMENSION_MISMATCH mask, w and/or u dimensions are incompatible.

6279 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
6280 responding domains of the accumulation operator or index unary
6281 operator, or the mask's domain is not compatible with bool (in
6282 the case where desc[GrB_MASK].GrB_STRUCTURE is not set).

6283 GrB_EMPTY_OBJECT The GrB_Scalar s used in the call is empty ($\mathbf{nvals}(s) = 0$) and
6284 therefore a value cannot be passed to the index unary operator.

6285 Description

6286 This variant of GrB_apply computes the result of applying an index unary operator to the elements
6287 of a GraphBLAS vector each composed with the element's index and a scalar constant, val or s:

$$6288 \quad \mathbf{w} = f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \text{val}) \text{ or } \mathbf{w} = f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}),$$

6289 or if an optional binary accumulation operator (\odot) is provided:

$$6290 \quad \mathbf{w} = \mathbf{w} \odot f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \text{val}) \text{ or } \mathbf{w} = \mathbf{w} \odot f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}).$$

6291 Logically, this operation occurs in three steps:

6292 **Setup** The internal vectors and mask used in the computation are formed and their domains
6293 and dimensions are tested for compatibility.

6294 **Compute** The indicated computations are carried out.

6295 **Output** The result is written into the output vector, possibly under control of a mask.

6296 Up to three argument vectors are used in this GrB_apply operation:

- 6297 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 6298 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)

6299 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6300 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)
6301 are tested for domain compatibility as follows:

- 6302 1. If \mathbf{mask} is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
6303 must be from one of the pre-defined types of Table 3.2.
- 6304 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$ of the index unary operator.
- 6305 3. If \mathbf{accum} is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
6306 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of the index unary operator must be compatible
6307 with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accumulation operator.
- 6308 4. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the index unary operator.
- 6309 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
6310 the index unary operator.
- 6311 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
6312 unary operator.

6313 Two domains are compatible with each other if values from one domain can be cast to values in
6314 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
6315 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6316 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
6317 error listed above is returned.

6318 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
6319 denotes copy):

- 6320 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 6321 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument \mathbf{mask} as follows:
 - 6322 (a) If $\mathbf{mask} = \mathbf{GrB_NULL}$, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 6323 (b) If $\mathbf{mask} \neq \mathbf{GrB_NULL}$,
 - 6324 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 6325 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 6326 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 6327 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 6328 4. Scalar $\tilde{s} \leftarrow \mathbf{s}$ (GraphBLAS scalar case).

6329 The internal vectors and masks are checked for dimension compatibility. The following conditions
6330 must hold:

6331 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

6332 2. $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$.

6333 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
6334 error listed above is returned.

6335 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6336 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6337 If an empty `GrB_Scalar` \tilde{s} is provided ($\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6338 If a non-empty `GrB_Scalar`, \tilde{s} , is provided ($\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable `val`
6339 with the same domain as \tilde{s} and set `val = val(\tilde{s})`.

6340 We are now ready to carry out the apply and any additional associated operations. We describe
6341 this in terms of two intermediate vectors:

- 6342 • $\tilde{\mathbf{t}}$: The vector holding the result from applying the index unary operator to the input vector
6343 $\tilde{\mathbf{u}}$.
- 6344 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

6345 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$6346 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f_i(\tilde{\mathbf{u}}(i), [i], 0, \text{val})) \mid i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,$$

6347 where $f_i = \mathbf{f}(\text{op})$.

6348 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 6349 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 6350 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$6351 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6352 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
6353 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 6354 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 6355 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6356 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

6359 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

6360 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
6361 using what is called a *standard vector mask and replace*. This is carried out under control of the
6362 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.

6375 4.3.8.6 apply: Matrix index unary operator variant[Scott: NEW CONTENT]

6376 Computes the transformation of the values of the stored elements of a matrix using an index unary
6377 operator that is a function of the stored value, its location indices, and an user provided scalar
6378 value. The scalar can be passed either as a non-opaque variable or as a GrB_Scalar object.

6379 C Syntax

```
6380     GrB_Info GrB_apply(GrB_Matrix      C,
6381                       const GrB_Matrix Mask,
6382                       const GrB_BinaryOp accum,
6383                       const GrB_IndexUnaryOp op,
6384                       const GrB_Matrix A,
6385                       <type>          val,
6386                       const GrB_Descriptor desc);
```

```
6387     GrB_Info GrB_apply(GrB_Matrix      C,
6388                       const GrB_Matrix Mask,
6389                       const GrB_BinaryOp accum,
6390                       const GrB_IndexUnaryOp op,
6391                       const GrB_Matrix A,
6392                       const GrB_Scalar s,
6393                       const GrB_Descriptor desc);
```

6394 Parameters

6395 C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
6396 that may be accumulated with the result of the apply operation. On output, the
6397 matrix holds the results of the operation.

6398 Mask (IN) An optional “write” mask that controls which results from this operation are
6399 stored into the output matrix C. The mask dimensions must match those of the
6400 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
6401 of the Mask matrix must be of type **bool** or any of the predefined “built-in” types
6402 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
6403 dimensions of C), GrB_NULL should be specified.

6404 accum (IN) An optional binary operator used for accumulating entries into existing C
6405 entries. If assignment rather than accumulation is desired, GrB_NULL should be
6406 specified.

6407 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
6408 to each element stored in the input matrix, A. It is a function of the stored element’s
6409 value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6410 A (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-
6411 ator.

6412 val (IN) An additional scalar value that is passed to the index unary operator.

6413 s (IN) An additional GraphBLAS scalar that is passed to the index unary operator.

6414 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
6415 should be specified. Non-default field/value pairs are listed as follows:
6416

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.

6418 Return Values

6419 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
6420 blocking mode, this indicates that the compatibility tests on di-
6421 mensions and domains for the input arguments passed successfully.
6422 Either way, output matrix C is ready to be used in the next method
6423 of the sequence.

6424 GrB_PANIC Unknown internal error.

6425 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
6426 GraphBLAS objects (input or output) is in an invalid state caused

6427 by a previous execution error. Call `GrB_error()` to access any error
 6428 messages generated by the implementation.

6429 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

6430 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 6431 a call to `new` (or another constructor).

6432 **GrB_DIMENSION_MISMATCH** `mask`, `w` and/or `u` dimensions are incompatible.

6433 **GrB_DOMAIN_MISMATCH** The domains of the various matrices are incompatible with the
 6434 corresponding domains of the accumulation operator or index unary
 6435 operator, or the mask's domain is not compatible with `bool` (in the
 6436 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6437 **GrB_EMPTY_OBJECT** The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and
 6438 therefore a value cannot be passed to the index unary operator.

6439 Description

6440 This variant of `GrB_apply` computes the result of applying a index unary operator to the elements
 6441 of a GraphBLAS matrix each composed with the elements row and column indices, and a scalar
 6442 constant, `val` or `s`:

$$6443 \quad C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s),$$

6444 or if an optional binary accumulation operator (\odot) is provided:

$$6445 \quad C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s).$$

6446 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional
 6447 indices, respectively.

6448 Logically, this operation occurs in three steps:

6449 **Setup** The internal matrices and mask used in the computation are formed and their domains
 6450 and dimensions are tested for compatibility.

6451 **Compute** The indicated computations are carried out.

6452 **Output** The result is written into the output matrix, possibly under control of a mask.

6453 Up to three argument matrices are used in the `GrB_apply` operation:

- 6454 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6455 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)

6456 3. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

6457 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)
6458 are tested for domain compatibility as follows:

- 6459 1. If **Mask** is not **GrB_NULL**, and **desc[GrB_MASK].GrB_STRUCTURE** is not set, then $\mathbf{D}(\mathbf{Mask})$
6460 must be from one of the pre-defined types of Table 3.2.
- 6461 2. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$ of the index unary operator.
- 6462 3. If **accum** is not **GrB_NULL**, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
6463 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of the index unary operator must be compatible
6464 with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accumulation operator.
- 6465 4. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the index unary operator.
- 6466 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
6467 the index unary operator.
- 6468 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
6469 unary operator.

6470 Two domains are compatible with each other if values from one domain can be cast to values in
6471 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
6472 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6473 any compatibility rule above is violated, execution of **GrB_apply** ends and the domain mismatch
6474 error listed above is returned.

6475 From the argument matrices, the internal matrices, **mask**, and index arrays used in the computation
6476 are formed (\leftarrow denotes copy):

- 6477 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 6478 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument **Mask** as follows:
 - 6479 (a) If **Mask** = **GrB_NULL**, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
6480 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 6481 (b) If **Mask** \neq **GrB_NULL**,
 - 6482 i. If **desc[GrB_MASK].GrB_STRUCTURE** is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
6483 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 6484 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
6485 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 6486 (c) If **desc[GrB_MASK].GrB_COMP** is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 6487 3. Matrix $\tilde{\mathbf{A}}$ is computed from argument **A** as follows:

$$6488 \quad \tilde{\mathbf{A}} \leftarrow \mathbf{desc[GrB_INP0].GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$$
- 6489 4. Scalar $\tilde{s} \leftarrow s$ (GraphBLAS scalar case).

6490 The internal matrices and mask are checked for dimension compatibility. The following conditions
6491 must hold:

- 6492 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 6493 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 6494 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
- 6495 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

6496 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
6497 error listed above is returned.

6498 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6499 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6500 If an empty `GrB_Scalar` \tilde{s} is provided ($\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6501 If a non-empty `GrB_Scalar`, \tilde{s} , is provided (i.e., $\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable
6502 \mathbf{val} with the same domain as \tilde{s} and set $\mathbf{val} = \mathbf{val}(\tilde{s})$.

6503 We are now ready to carry out the apply and any additional associated operations. We describe
6504 this in terms of two intermediate matrices:

- 6505 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the index unary operator to the input matrix
6506 $\tilde{\mathbf{A}}$.
- 6507 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

6508 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$6509 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f_i(\tilde{\mathbf{A}}(i, j), i, j, \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

6510 where $f_i = \mathbf{f}(\mathbf{op})$.

6511 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 6512 • If $\mathbf{accum} = \mathbf{GrB_NULL}$, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 6513 • If \mathbf{accum} is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$6514 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6515 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
6516 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 6517 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6518 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6519 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6520 \quad & \\ 6521 \end{aligned}$$

6522 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

6523 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
6524 using what is called a *standard matrix mask and replace*. This is carried out under control of the
6525 mask which acts as a “write mask”.

- 6526 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
6527 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$6528 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6529 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
6530 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
6531 mask are unchanged:

$$6532 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6533 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
6534 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
6535 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
6536 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
6537 sequence.

6538 4.3.9 select:

6539 Apply a select operator to the stored elements of an object to determine whether or not to keep
6540 them.

6541 4.3.9.1 select: Vector variant[Scott: NEW CONTENT]

6542 Apply a select operator (an index unary operator) to the elements of a vector.

6543 C Syntax

```
6544 // scalar value variant
6545 GrB_Info GrB_select(GrB_Vector          w,
6546                    const GrB_Vector      mask,
6547                    const GrB_BinaryOp    accum,
6548                    const GrB_IndexUnaryOp op,
6549                    const GrB_Vector      u,
6550                    <type>                val,
6551                    const GrB_Descriptor   desc);
6552
6553 // GraphBLAS scalar variant
6554 GrB_Info GrB_select(GrB_Vector          w,
6555                    const GrB_Vector      mask,
```

```

6556         const GrB_BinaryOp      accum,
6557         const GrB_IndexUnaryOp  op,
6558         const GrB_Vector        u,
6559         const GrB_Scalar        s,
6560         const GrB_Descriptor    desc);
6561

```

6562 Parameters

6563 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
6564 that may be accumulated with the result of the select operation. On output, this
6565 vector holds the results of the operation.

6566 **mask** (IN) An optional “write” mask that controls which results from this operation are
6567 stored into the output vector **w**. The mask dimensions must match those of the
6568 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
6569 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
6570 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
6571 dimensions of **w**), **GrB_NULL** should be specified.

6572 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
6573 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
6574 specified.

6575 **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
6576 to each element stored in the input vector, **u**. It is a function of the stored element’s
6577 value, its location index, and a user supplied scalar value (either **s** or **val**).

6578 **u** (IN) The GraphBLAS vector whose elements are passed to the index unary oper-
6579 ator.

6580 **val** (IN) An additional scalar value that is passed to the index unary operator.

6581 **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must
6582 not be empty.

6583 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
6584 should be specified. Non-default field/value pairs are listed as follows:

6585

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 .

6586

6587 Return Values

6588 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 6589 blocking mode, this indicates that the compatibility tests on di-
 6590 mensions and domains for the input arguments passed success-
 6591 fully. Either way, output vector **w** is ready to be used in the next
 6592 method of the sequence.

6593 GrB_PANIC Unknown internal error.

6594 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
 6595 opaque GraphBLAS objects (input or output) is in an invalid
 6596 state caused by a previous execution error. Call **GrB_error()** to
 6597 access any error messages generated by the implementation.

6598 GrB_OUT_OF_MEMORY Not enough memory available for operation.

6599 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
 6600 by a call to one of its constructors.

6601 GrB_DIMENSION_MISMATCH **mask**, **w** and/or **u** dimensions are incompatible.

6602 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
 6603 responding domains of the accumulation operator or index unary
 6604 operator, or the **mask**'s domain is not compatible with **bool** (in
 6605 the case where **desc[GrB_MASK].GrB_STRUCTURE** is not set).

6606 GrB_EMPTY_OBJECT The **GrB_Scalar s** used in the call is empty (**nvals(s) = 0**) and
 6607 therefore a value cannot be passed to the index unary operator.

6608 Description

6609 This variant of **GrB_select** computes the result of applying a index unary operator to select the
 6610 elements of the input GraphBLAS vector. The operator takes, as input, the value of each stored
 6611 element, along with the element's index and a scalar constant – either **val** or **s**. The corresponding
 6612 element of the input vector is selected (kept) if the function evaluates to **true** when cast to **bool**.
 6613 This acts like a functional mask on the input vector as follows:

$$6614 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle,$$

$$6615 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle.$$

6616 Correspondingly, if a **GrB_Scalar s**, is provided:

$$6617 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle,$$

$$6618 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle.$$

6619 Logically, this operation occurs in three steps:

6620 **Setup** The internal vectors and mask used in the computation are formed and their domains
6621 and dimensions are tested for compatibility.

6622 **Compute** The indicated computations are carried out.

6623 **Output** The result is written into the output vector, possibly under control of a mask.

6624 Up to three argument vectors are used in this `GrB_select` operation:

- 6625 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
6626 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
6627 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6628 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)
6629 are tested for domain compatibility as follows:

- 6630 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
6631 must be from one of the pre-defined types of Table 3.2.
6632 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}(\mathbf{u})$.
6633 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
6634 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accu-
6635 mulation operator.
6636 4. $\mathbf{D}_{out}(\mathbf{op})$ of the index unary operator must be from one of the pre-defined types of Table 3.2;
6637 i.e., castable to `bool`.
6638 5. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the index unary operator.
6639 6. $\mathbf{D}(\mathbf{val})$ or $\mathbf{D}(\mathbf{s})$, depending on the signature of the method, must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$
6640 of the index unary operator.

6641 Two domains are compatible with each other if values from one domain can be cast to values in
6642 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
6643 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6644 any compatibility rule above is violated, execution of `GrB_select` ends and the domain mismatch
6645 error listed above is returned.

6646 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
6647 denotes copy):

- 6648 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
6649 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:

6650 (a) If $\text{mask} = \text{GrB_NULL}$, then $\widetilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
6651 (b) If $\text{mask} \neq \text{GrB_NULL}$,
6652 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
6653 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
6654 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
6655 3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$.
6656 4. Scalar $\widetilde{s} \leftarrow s$ (GrB_Scalar version only).

6657 The internal vectors and masks are checked for dimension compatibility. The following conditions
6658 must hold:

- 6659 1. $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 6660 2. $\text{size}(\widetilde{\mathbf{u}}) = \text{size}(\widetilde{\mathbf{w}})$.

6661 If any compatibility rule above is violated, execution of `GrB_select` ends and the dimension mismatch
6662 error listed above is returned.

6663 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6664 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6665 If an empty `GrB_Scalar` \widetilde{s} is provided (i.e., $\text{nvals}(\widetilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6666 If a non-empty `GrB_Scalar`, \widetilde{s} , is provided (i.e., $\text{nvals}(\widetilde{s}) = 1$), we then create an internal variable
6667 `val` with the same domain as \widetilde{s} and set $\text{val} = \text{val}(\widetilde{s})$.

6668 We are now ready to carry out the `select` and any additional associated operations. We describe
6669 this in terms of two intermediate vectors:

- 6670 • $\widetilde{\mathbf{t}}$: The vector holding the result from applying the index unary operator to the input vector
6671 $\widetilde{\mathbf{u}}$.
- 6672 • $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

6673 The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$6674 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{u}}), \{(i, \widetilde{\mathbf{u}}(i), : i \in \text{ind}(\widetilde{\mathbf{u}}) \wedge (\text{bool})f_i(\widetilde{\mathbf{u}}(i), i, 0, \text{val}) = \text{true})\} \rangle,$$

6675 where $f_i = \mathbf{f}(\text{op})$.

6676 The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 6677 • If $\text{accum} = \text{GrB_NULL}$, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.
- 6678 • If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$6679 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

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);
```

```

6715 // GraphBLAS scalar variant
6716 GrB_Info GrB_select(GrB_Matrix          C,
6717                    const GrB_Matrix     Mask,
6718                    const GrB_BinaryOp   accum,
6719                    const GrB_IndexUnaryOp op,
6720                    const GrB_Matrix     A,
6721                    const GrB_Scalar     s,
6722                    const GrB_Descriptor desc);

```

6723 Parameters

6724 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
6725 that may be accumulated with the result of the select operation. On output, the
6726 matrix holds the results of the operation.

6727 **Mask** (IN) An optional “write” mask that controls which results from this operation are
6728 stored into the output matrix **C**. The mask dimensions must match those of the
6729 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
6730 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
6731 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
6732 dimensions of **C**), **GrB_NULL** should be specified.

6733 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
6734 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
6735 specified.

6736 **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
6737 to each element stored in the input matrix, **A**. It is a function of the stored element’s
6738 value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6739 **A** (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-
6740 ator.

6741 **val** (IN) An additional scalar value that is passed to the index unary operator.

6742 **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must
6743 not be empty.

6744 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
6745 should be specified. Non-default field/value pairs are listed as follows:

6746

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:

6776 $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$, or
6777 $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$.

6778 Correspondingly, if a GrB_Scalar, *s*, is provided:

6779 $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$, or
6780 $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$.

6781 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional
6782 indices, respectively.

6783 Logically, this operation occurs in three steps:

6784 **Setup** The internal matrices and mask used in the computation are formed and their domains
6785 and dimensions are tested for compatibility.

6786 **Compute** The indicated computations are carried out.

6787 **Output** The result is written into the output matrix, possibly under control of a mask.

6788 Up to three argument matrices are used in the GrB_select operation:

- 6789 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6790 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)
- 6791 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6792 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)
6793 are tested for domain compatibility as follows:

- 6794 1. If **Mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{Mask})$
6795 must be from one of the pre-defined types of Table 3.2.
- 6796 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.
- 6797 3. If **accum** is not GrB_NULL, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
6798 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
6799 mulation operator.
- 6800 4. $\mathbf{D}_{out}(\text{op})$ of the index unary operator must be from one of the pre-defined types of Table 3.2;
6801 i.e., castable to **bool**.
- 6802 5. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the index unary operator.
- 6803 6. $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$, depending on the signature of the method, must be compatible with $\mathbf{D}_{in_2}(\text{op})$
6804 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:

- 6839 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the index unary operator to the input matrix
6840 $\tilde{\mathbf{A}}$.
- 6841 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

6842 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$6843 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \\ \{(i, j, \tilde{\mathbf{A}}(i, j) : i, j \in \mathbf{ind}(\tilde{\mathbf{A}}) \wedge (\text{bool})f_i(\tilde{\mathbf{A}}(i, j), i, j, \text{val}) = \text{true})\},$$

6844 where $f_i = \mathbf{f}(\text{op})$.

6845 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 6846 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 6847 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$6848 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6849 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
6850 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$6851 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6852 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6853 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6854 \quad 6855$$

6856 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

6857 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
6858 using what is called a *standard matrix mask and replace*. This is carried out under control of the
6859 mask which acts as a “write mask”.

- 6860 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
6861 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$6862 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6863 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
6864 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
6865 mask are unchanged:

$$6866 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6867 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
6868 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
6869 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
6870 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
6871 sequence.

6872 4.3.10 reduce: Perform a reduction across the elements of an object

6873 Computes the reduction of the values of the elements of a vector or matrix.

6874 4.3.10.1 reduce: Standard matrix to vector variant

6875 This performs a reduction across rows of a matrix to produce a vector. If reduction down columns
6876 is desired, the input matrix should be transposed using the descriptor.

6877 C Syntax

```
6878     GrB_Info GrB_reduce(GrB_Vector      w,  
6879                        const GrB_Vector mask,  
6880                        const GrB_BinaryOp accum,  
6881                        const GrB_Monoid op,  
6882                        const GrB_Matrix A,  
6883                        const GrB_Descriptor desc);  
6884  
6885     GrB_Info GrB_reduce(GrB_Vector      w,  
6886                        const GrB_Vector mask,  
6887                        const GrB_BinaryOp accum,  
6888                        const GrB_BinaryOp op,  
6889                        const GrB_Matrix A,  
6890                        const GrB_Descriptor desc);
```

6891 Parameters

6892 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
6893 that may be accumulated with the result of the reduction operation. On output,
6894 this vector holds the results of the operation.

6895 **mask** (IN) An optional “write” mask that controls which results from this operation are
6896 stored into the output vector **w**. The mask dimensions must match those of the
6897 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
6898 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
6899 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
6900 dimensions of **w**), **GrB_NULL** should be specified.

6901 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
6902 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
6903 specified.

6904 **op** (IN) The monoid or binary operator used in the element-wise reduction operation.
6905 Depending on which type is passed, the following defines the binary operator with
6906 one domain, $F_b = \langle D, D, D, \oplus \rangle$, that is used:

6907 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.
6908 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
6909 monoid is ignored.

6910 If `op` is a `GrB_BinaryOp`, then all its domains must be the same. Furthermore, in
6911 both cases $\odot(\text{op})$ must be commutative and associative. Otherwise, the outcome
6912 of the operation is undefined.

6913 **A** (IN) The GraphBLAS matrix on which reduction will be performed.

6914 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
6915 should be specified. Non-default field/value pairs are listed as follows:
6916

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.

6918 Return Values

6919 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
6920 blocking mode, this indicates that the compatibility tests on di-
6921 mensions and domains for the input arguments passed successfully.
6922 Either way, output vector `w` is ready to be used in the next method
6923 of the sequence.

6924 **GrB_PANIC** Unknown internal error.

6925 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
6926 GraphBLAS objects (input or output) is in an invalid state caused
6927 by a previous execution error. Call `GrB_error()` to access any error
6928 messages generated by the implementation.

6929 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

6930 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
6931 a call to `new` (or `dup` for vector parameters).

6932 **GrB_DIMENSION_MISMATCH** `mask`, `w` and/or `u` dimensions are incompatible.

6933 **GrB_DOMAIN_MISMATCH** Either the domains of the various vectors and matrices are incom-
6934 patible with the corresponding domains of the accumulation oper-
6935 ator or reduce function, or the domains of the GraphBLAS binary

operator `op` are not all the same, or the mask's domain is not compatible with `bool` (in the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6939 Description

6940 This variant of `GrB_reduce` computes the result of performing a reduction across each of the rows
 6941 of an input matrix: $w(i) = \bigoplus A(i, :) \forall i$; or, if an optional binary accumulation operator is provided,
 6942 $w(i) = w(i) \odot (\bigoplus A(i, :)) \forall i$, where $\bigoplus = \odot(F_b)$ and $\odot = \odot(\text{accum})$.

6943 Logically, this operation occurs in three steps:

6944 **Setup** The internal vector, matrix and mask used in the computation are formed and their
 6945 domains and dimensions are tested for compatibility.

6946 **Compute** The indicated computations are carried out.

6947 **Output** The result is written into the output vector, possibly under control of a mask.

6948 Up to two vector and one matrix argument are used in this `GrB_reduce` operation:

- 6949 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6950 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 6951 3. $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6952 The argument vector, matrix, reduction operator and accumulation operator (if provided) are tested
 6953 for domain compatibility as follows:

- 6954 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
 6955 must be from one of the pre-defined types of Table 3.2.
- 6956 2. $\mathbf{D}(w)$ must be compatible with the domain of the reduction binary operator, $\mathbf{D}(F_b)$.
- 6957 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 6958 of the accumulation operator and $\mathbf{D}(F_b)$, must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 6959 mulation operator.
- 6960 4. $\mathbf{D}(A)$ must be compatible with the domain of the binary reduction operator, $\mathbf{D}(F_b)$.

6961 Two domains are compatible with each other if values from one domain can be cast to values in
 6962 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 6963 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 6964 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch
 6965 error listed above is returned.

6966 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 6967 denotes copy):

- 6968 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 6969 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
- 6970 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
- 6971 (b) If `mask \neq GrB_NULL`,
- 6972 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
- 6973 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
- 6974 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 6975 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

6976 The internal vectors and masks are checked for dimension compatibility. The following conditions
 6977 must hold:

- 6978 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 6979 2. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.

6980 If any compatibility rule above is violated, execution of `GrB_reduce` ends and the dimension mis-
 6981 match error listed above is returned.

6982 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 6983 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6984 We carry out the reduce and any additional associated operations. We describe this in terms of
 6985 two intermediate vectors:

- 6986 • $\tilde{\mathbf{t}}$: The vector holding the result from reducing along the rows of input matrix $\tilde{\mathbf{A}}$.
- 6987 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

6988 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$6989 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, t_i) : \mathbf{ind}(\mathbf{A}(i, :)) \neq \emptyset\} \rangle.$$

6990 The value of each of its elements is computed by

$$6991 \quad t_i = \bigoplus_{j \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :))} \tilde{\mathbf{A}}(i, j),$$

6992 where $\bigoplus = \odot(F_b)$.

6993 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 6994 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

6995 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$6996 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6997 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
6998 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 6999 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 7000 \\ 7001 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 7002 \\ 7003 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

7004 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

7005 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
7006 using what is called a *standard vector mask and replace*. This is carried out under control of the
7007 mask which acts as a “write mask”.

7008 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
7009 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$7010 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

7011 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
7012 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
7013 mask are unchanged:

$$7014 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

7015 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
7016 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
7017 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
7018 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
7019 sequence.

7020 4.3.10.2 reduce: Vector-scalar variant[Scott: NEW CONTENT]

7021 Reduce all stored values into a single scalar.

7022 C Syntax

```
7023 // scalar value + monoid (only)
7024 GrB_Info GrB_reduce(<type>          *val,
7025                      const GrB_BinaryOp accum,
7026                      const GrB_Monoid  op,
7027                      const GrB_Vector  u,
```

```

7028             const GrB_Descriptor desc);
7029
7030 // GraphBLAS Scalar + monoid
7031 GrB_Info GrB_reduce(GrB_Scalar      s,
7032                   const GrB_BinaryOp accum,
7033                   const GrB_Monoid  op,
7034                   const GrB_Vector  u,
7035                   const GrB_Descriptor desc);
7036
7037 // GraphBLAS Scalar + binary operator
7038 GrB_Info GrB_reduce(GrB_Scalar      s,
7039                   const GrB_BinaryOp accum,
7040                   const GrB_BinaryOp op,
7041                   const GrB_Vector  u,
7042                   const GrB_Descriptor desc);

```

7043 Parameters

7044 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides
7045 a value that may be accumulated (optionally) with the result of the reduction
7046 operation. On output, this scalar holds the results of the operation.

7047 **accum** (IN) An optional binary operator used for accumulating entries into an exist-
7048 ing scalar (**s** or **val**) value. If assignment rather than accumulation is desired,
7049 **GrB_NULL** should be specified.

7050 **op** (IN) The monoid ($M = \langle D, \oplus, 0 \rangle$) or binary operator ($F_b = \langle D, D, D, \oplus \rangle$) used in
7051 the reduction operation. The \oplus operator must be commutative and associative;
7052 otherwise, the outcome of the operation is undefined.

7053 **u** (IN) The GraphBLAS vector on which reduction will be performed.

7054 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
7055 should be specified. Non-default field/value pairs are listed as follows:

7057 Param	Field	Value	Description
------------	-------	-------	-------------

7058 *Note:* This argument is defined for consistency with the other GraphBLAS opera-
7059 tions. There are currently no non-default field/value pairs that can be set for this
7060 operation.

7061 Return Values

7062 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
7063 cessfully, and the output scalar (**s** or **val**) is ready to be used in the
7064 next method of the sequence.

7065 GrB_PANIC Unknown internal error.

7066 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
7067 GraphBLAS objects (input or output) is in an invalid state caused
7068 by a previous execution error. Call GrB_error() to access any error
7069 messages generated by the implementation.

7070 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

7071 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
7072 a call to a respective constructor.

7073 GrB_NULL_POINTER val pointer is NULL.

7074 GrB_DOMAIN_MISMATCH The domains of input and output arguments are incompatible with
7075 the corresponding domains of the accumulation operator, or reduce
7076 operator.

7077 Description

7078 This variant of GrB_reduce computes the result of performing a reduction across all of the stored
7079 elements of an input vector storing the result into either s or val. This corresponds to (shown here
7080 for the scalar value case only):

$$7081 \quad \text{val} = \begin{cases} \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i), & \text{or} \\ \text{val} \odot \left[\bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7082 where $\bigoplus = \odot(\text{op})$ and $\odot = \odot(\text{accum})$.

7083 Logically, this operation occurs in three steps:

7084 **Setup** The internal vector used in the computation is formed and its domain is tested for
7085 compatibility.

7086 **Compute** The indicated computations are carried out.

7087 **Output** The result is written into the output scalar.

7088 One vector argument is used in this GrB_reduce operation:

- 7089 1. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

7090 The output scalar, argument vector, reduction operator and accumulation operator (if provided)
7091 are tested for domain compatibility as follows:

- 7092 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
7093 $\mathbf{D}_{in_1}(\text{op})$ and $\mathbf{D}_{in_2}(\text{op})$ from F_b).

- 7094 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
7095 $\mathbf{D}_{out}(\text{accum})$ of the accumulation operator, and $\mathbf{D}(\text{op})$ from M (or $\mathbf{D}_{out}(\text{op})$ from F_b) must
7096 be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 7097 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).

7098 Two domains are compatible with each other if values from one domain can be cast to values in
7099 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
7100 compatible with each other. A domain from a user-defined type is only compatible with itself. If
7101 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch
7102 error listed above is returned.

7103 The number of values stored in the input, `u`, is checked. If there are no stored values in `u`, then one
7104 of the following occurs depending on the output variant:

$$7105 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if } \text{accum} = \text{GrB_NULL}, \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7106 or

$$7107 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if } \text{accum} = \text{GrB_NULL}, \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7108 where $\mathbf{0}(\text{op})$ is the identity of the monoid. The operation returns immediately with `GrB_SUCCESS`.

7109 For all other cases, the internal vector and scalar used in the computation is formed (\leftarrow denotes
7110 copy):

- 7111 1. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 7112 2. Scalar $\tilde{s} \leftarrow \text{s}$ (GraphBLAS scalar case).

7113 We are now ready to carry out the reduction and any additional associated operations. An inter-
7114 mediate scalar result t is computed as follows:

$$7115 \quad t = \bigoplus_{i \in \text{ind}(\tilde{\mathbf{u}})} \tilde{\mathbf{u}}(i),$$

7116 where $\oplus = \odot(\text{op})$.

7117 The final reduction value is computed as follows:

$$7118 \quad \mathbf{L}(\text{s}) \leftarrow \begin{cases} \{t\}, & \text{when } \text{accum} = \text{GrB_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\text{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7119 or

$$7120 \quad \text{val} \leftarrow \begin{cases} t, & \text{when } \text{accum} = \text{GrB_NULL, or} \\ \text{val} \odot t, & \text{otherwise;} \end{cases}$$

7121 In both GrB_BLOCKING and GrB_NONBLOCKING modes, the method exits with return value
7122 GrB_SUCCESS and the new contents of the output scalar is as defined above.

7123 4.3.10.3 reduce: Matrix-scalar variant[Scott: NEW CONTENT]

7124 Reduce all stored values into a single scalar.

7125 C Syntax

```
7126 // scalar value + monoid (only)
7127 GrB_Info GrB_reduce(<type>          *val,
7128                   const GrB_BinaryOp accum,
7129                   const GrB_Monoid   op,
7130                   const GrB_Matrix   A,
7131                   const GrB_Descriptor desc);
7132
7133 // GraphBLAS Scalar + monoid
7134 GrB_Info GrB_reduce(GrB_Scalar      s,
7135                   const GrB_BinaryOp accum,
7136                   const GrB_Monoid   op,
7137                   const GrB_Matrix   A,
7138                   const GrB_Descriptor desc);
7139
7140 // GraphBLAS Scalar + binary operator
7141 GrB_Info GrB_reduce(GrB_Scalar      s,
7142                   const GrB_BinaryOp accum,
7143                   const GrB_BinaryOp op,
7144                   const GrB_Matrix   A,
7145                   const GrB_Descriptor desc);
```

7146 Parameters

7147 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides
7148 a value that may be accumulated (optionally) with the result of the reduction
7149 operation. On output, this scalar holds the results of the operation.

7150 **accum** (IN) An optional binary operator used for accumulating entries into existing (**s** or
7151 **val**) value. If assignment rather than accumulation is desired, GrB_NULL should
7152 be specified.

7153 **op** (IN) The monoid ($M = \langle D, \oplus, 0 \rangle$) or binary operator ($F_b = \langle D, D, D, \oplus \rangle$) used in
7154 the reduction operation. The \oplus operator must be commutative and associative;
7155 otherwise, the outcome of the operation is undefined.

7156 **A** (IN) The GraphBLAS matrix on which the reduction will be performed.

7157 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 7158 should be specified. Non-default field/value pairs are listed as follows:

7159

Param	Field	Value	Description
-------	-------	-------	-------------

7161 *Note:* This argument is defined for consistency with the other GraphBLAS opera-
 7162 tions. There are currently no non-default field/value pairs that can be set for this
 7163 operation.

7164 Return Values

7165 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
 7166 cessfully, and the output scalar (s or val) is ready to be used in the
 7167 next method of the sequence.

7168 GrB_PANIC Unknown internal error.

7169 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 7170 GraphBLAS objects (input or output) is in an invalid state caused
 7171 by a previous execution error. Call GrB_error() to access any error
 7172 messages generated by the implementation.

7173 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

7174 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 7175 a call to a respective constructor.

7176 GrB_NULL_POINTER val pointer is NULL.

7177 GrB_DOMAIN_MISMATCH The domains of input and output arguments are incompatible with
 7178 the corresponding domains of the accumulation operator, or reduce
 7179 operator.

7180 Description

7181 This variant of GrB_reduce computes the result of performing a reduction across all of the stored
 7182 elements of an input matrix storing the result into either s or val. This corresponds to (shown here
 7183 for the scalar value case only):

$$7184 \quad \text{val} = \begin{cases} \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j), & \text{or} \\ \text{val} \odot \left[\bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7185 where $\bigoplus = \odot(\text{op})$ and $\odot = \odot(\text{accum})$.

7186 Logically, this operation occurs in three steps:

7187 **Setup** The internal matrix used in the computation is formed and its domain is tested for
 7188 compatibility.

7189 **Compute** The indicated computations are carried out.

7190 **Output** The result is written into the output scalar.

7191 One matrix argument is used in this GrB_reduce operation:

7192 1. $A = \langle \mathbf{D}(A), \mathbf{size}(A), \mathbf{L}(A) = \{(i, j, A_{i,j})\} \rangle$

7193 The output scalar, argument matrix, reduction operator and accumulation operator (if provided)
 7194 are tested for domain compatibility as follows:

7195 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
 7196 $\mathbf{D}_{in_1}(\text{op})$ and $\mathbf{D}_{in_2}(\text{op})$ from F_b).

7197 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
 7198 $\mathbf{D}_{out}(\text{accum})$ of the accumulation operator, and $\mathbf{D}(\text{op})$ from M (or $\mathbf{D}_{out}(\text{op})$ from F_b) must
 7199 be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

7200 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).

7201 Two domains are compatible with each other if values from one domain can be cast to values in
 7202 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 7203 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 7204 any compatibility rule above is violated, execution of GrB_reduce ends and the domain mismatch
 7205 error listed above is returned.

7206 The number of values stored in the input, A , is checked. If there are no stored values in A , then
 7207 one of the following occurs depending on the output variant:

$$7208 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if accum = GrB_NULL,} \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7209 or

$$7210 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if accum = GrB_NULL,} \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7211 where $\mathbf{0}(\text{op})$ is the identity of the monoid. The operation returns immediately with GrB_SUCCESS.

7212 For all other cases, the internal matrix and scalar used in the computation is formed (\leftarrow denotes
 7213 copy):

7214 1. Matrix $\tilde{A} \leftarrow A$.

7215 2. Scalar $\tilde{s} \leftarrow s$ (GraphBLAS scalar case).

7216 We are now ready to carry out the reduce and any additional associated operations. An intermediate
 7217 scalar result t is computed as follows:

$$7218 \quad t = \bigoplus_{(i,j) \in \text{ind}(\tilde{\mathbf{A}})} \tilde{\mathbf{A}}(i,j),$$

7219 where $\oplus = \odot(\text{op})$.

7220 The final reduction value is computed as follows:

$$7221 \quad \mathbf{L}(\mathbf{s}) \leftarrow \begin{cases} \{t\}, & \text{when accum} = \text{GrB_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\mathbf{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7222 or

$$7223 \quad \mathbf{val} \leftarrow \begin{cases} t, & \text{when accum} = \text{GrB_NULL, or} \\ \mathbf{val} \odot t, & \text{otherwise;} \end{cases}$$

7224 In both GrB_BLOCKING and GrB_NONBLOCKING modes, the method exits with return value
 7225 GrB_SUCCESS and the new contents of the output scalar is as defined above.

7226 4.3.11 transpose: Transpose rows and columns of a matrix

7227 This version computes a new matrix that is the transpose of the source matrix.

7228 C Syntax

```
7229      GrB_Info GrB_transpose(GrB_Matrix      C,
7230                           const GrB_Matrix Mask,
7231                           const GrB_BinaryOp accum,
7232                           const GrB_Matrix A,
7233                           const GrB_Descriptor desc);
```

7234 Parameters

7235 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
 7236 that may be accumulated with the result of the transpose operation. On output,
 7237 the matrix holds the results of the operation.

7238 **Mask** (IN) An optional “write” mask that controls which results from this operation are
 7239 stored into the output matrix C. The mask dimensions must match those of the
 7240 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
 7241 of the Mask matrix must be of type bool or any of the predefined “built-in” types
 7242 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
 7243 dimensions of C), GrB_NULL should be specified.

7244 **accum** (IN) An optional binary operator used for accumulating entries into existing C
 7245 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
 7246 specified.

7247 **A** (IN) The GraphBLAS matrix on which transposition will be performed.

7248 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
 7249 should be specified. Non-default field/value pairs are listed as follows:

7250

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.

7251

7252 Return Values

7253 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 7254 blocking mode, this indicates that the compatibility tests on di-
 7255 mensions and domains for the input arguments passed successfully.
 7256 Either way, output matrix C is ready to be used in the next method
 7257 of the sequence.

7258 **GrB_PANIC** Unknown internal error.

7259 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 7260 GraphBLAS objects (input or output) is in an invalid state caused
 7261 by a previous execution error. Call **GrB_error()** to access any error
 7262 messages generated by the implementation.

7263 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

7264 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 7265 a call to **new** (or **Matrix_dup** for matrix parameters).

7266 **GrB_DIMENSION_MISMATCH** mask, C and/or A dimensions are incompatible.

7267 **GrB_DOMAIN_MISMATCH** The domains of the various matrices are incompatible with the cor-
 7268 responding domains of the accumulation operator, or the mask's do-
 7269 main is not compatible with **bool** (in the case where **desc[GrB_MASK].GrB_STRUCTURE**
 7270 is not set).

7271 **Description**

7272 GrB_transpose computes the result of performing a transpose of the input matrix: $C = A^T$; or, if an
 7273 optional binary accumulation operator (\odot) is provided, $C = C \odot A^T$. We note that the input matrix
 7274 A can itself be optionally transposed before the operation, which would cause either an assignment
 7275 from A to C or an accumulation of A into C.

7276 Logically, this operation occurs in three steps:

7277 **Setup** The internal matrix and mask used in the computation are formed and their domains
 7278 and dimensions are tested for compatibility.

7279 **Compute** The indicated computations are carried out.

7280 **Output** The result is written into the output matrix, possibly under control of a mask.

7281 Up to three matrix arguments are used in this GrB_transpose operation:

- 7282 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7283 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)
- 7284 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

7285 The argument matrices and accumulation operator (if provided) are tested for domain compatibility
 7286 as follows:

- 7287 1. If Mask is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{Mask})$
 7288 must be from one of the pre-defined types of Table 3.2.
- 7289 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$ of the input matrix.
- 7290 3. If accum is not GrB_NULL, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 7291 of the accumulation operator and $\mathbf{D}(A)$ of the input matrix must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
 7292 of the accumulation operator.

7293 Two domains are compatible with each other if values from one domain can be cast to values in
 7294 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 7295 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 7296 any compatibility rule above is violated, execution of GrB_transpose ends and the domain mismatch
 7297 error listed above is returned.

7298 From the argument matrices, the internal matrices and mask used in the computation are formed
 7299 (\leftarrow denotes copy):

- 7300 1. Matrix $\tilde{C} \leftarrow C$.
- 7301 2. Two-dimensional mask, \tilde{M} , is computed from argument Mask as follows:

- 7302 (a) If $\text{Mask} = \text{GrB_NULL}$, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
7303 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
- 7304 (b) If $\text{Mask} \neq \text{GrB_NULL}$,
- 7305 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
7306 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
- 7307 ii. Otherwise, $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
7308 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
- 7309 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{M}} \leftarrow \neg \widetilde{\mathbf{M}}$.
- 7310 3. Matrix $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

7311 The internal matrices and masks are checked for dimension compatibility. The following conditions
7312 must hold:

- 7313 1. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$.
- 7314 2. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$.
- 7315 3. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{A}})$.
- 7316 4. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$.

7317 If any compatibility rule above is violated, execution of `GrB_transpose` ends and the dimension
7318 mismatch error listed above is returned.

7319 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
7320 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

7321 We are now ready to carry out the matrix transposition and any additional associated operations.
7322 We describe this in terms of two intermediate matrices:

- 7323 • $\widetilde{\mathbf{T}}$: The matrix holding the transpose of $\widetilde{\mathbf{A}}$.
- 7324 • $\widetilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

7325 The intermediate matrix

$$7326 \quad \widetilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \{(j, i, A_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{A}})\} \rangle$$

7327 is created.

7328 The intermediate matrix $\widetilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 7329 • If $\text{accum} = \text{GrB_NULL}$, then $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$.
- 7330 • If accum is a binary operator, then $\widetilde{\mathbf{Z}}$ is defined as

$$7331 \quad \widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\} \rangle.$$

7332 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 7333 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned}
 7334 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\
 7335 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 7336 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 7337 \quad & \\
 7338 \quad &
 \end{aligned}$$

7339 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

7340 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 7341 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 7342 mask which acts as a “write mask”.

- 7343 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
 7344 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$7345 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7346 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 7347 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 7348 mask are unchanged:

$$7349 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7350 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 7351 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 7352 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
 7353 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 7354 sequence.

7355 4.3.12 kronecker: Kronecker product of two matrices

7356 Computes the Kronecker product of two matrices. The result is a matrix.

7357 C Syntax

```

7358      GrB_Info GrB_kronecker(GrB_Matrix      C,
7359                             const GrB_Matrix  Mask,
7360                             const GrB_BinaryOp accum,
7361                             const GrB_Semiring op,
7362                             const GrB_Matrix  A,
7363                             const GrB_Matrix  B,
7364                             const GrB_Descriptor desc);
7365  
```



```

7366 GrB_Info GrB_kronecker(GrB_Matrix      C,
7367                        const GrB_Matrix  Mask,
7368                        const GrB_BinaryOp accum,
7369                        const GrB_Monoid   op,
7370                        const GrB_Matrix  A,
7371                        const GrB_Matrix  B,
7372                        const GrB_Descriptor desc);
7373
7374 GrB_Info GrB_kronecker(GrB_Matrix      C,
7375                        const GrB_Matrix  Mask,
7376                        const GrB_BinaryOp accum,
7377                        const GrB_BinaryOp op,
7378                        const GrB_Matrix  A,
7379                        const GrB_Matrix  B,
7380                        const GrB_Descriptor desc);

```

7381 Parameters

7382 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
7383 that may be accumulated with the result of the Kronecker product. On output,
7384 the matrix holds the results of the operation.

7385 **Mask** (IN) An optional “write” mask that controls which results from this operation are
7386 stored into the output matrix C. The mask dimensions must match those of the
7387 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
7388 of the Mask matrix must be of type bool or any of the predefined “built-in” types
7389 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
7390 dimensions of C), GrB_NULL should be specified.

7391 **accum** (IN) An optional binary operator used for accumulating entries into existing C
7392 entries. If assignment rather than accumulation is desired, GrB_NULL should be
7393 specified.

7394 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
7395 operation. Depending on which type is passed, the following defines the binary
7396 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

7397 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

7398 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-
7399 nored.

7400 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
7401 is ignored.

7402 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
7403 product.

7404 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
7405 product.

7406 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
7407 should be specified. Non-default field/value pairs are listed as follows:
7408

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.

7410 Return Values

7411 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
7412 blocking mode, this indicates that the compatibility tests on di-
7413 mensions and domains for the input arguments passed successfully.
7414 Either way, output matrix C is ready to be used in the next method
7415 of the sequence.

7416 GrB_PANIC Unknown internal error.

7417 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
7418 GraphBLAS objects (input or output) is in an invalid state caused
7419 by a previous execution error. Call GrB_error() to access any error
7420 messages generated by the implementation.

7421 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

7422 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
7423 a call to new (or Matrix_dup for matrix parameters).

7424 GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

7425 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
7426 corresponding domains of the binary operator (op) or accumulation
7427 operator, or the mask's domain is not compatible with bool (in the
7428 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

7429 Description

7430 GrB_kronecker computes the Kronecker product $C = A \otimes B$ or, if an optional binary accumulation
7431 operator (\odot) is provided, $C = C \odot (A \otimes B)$ (where matrices A and B can be optionally transposed).

7432 The Kronecker product is defined as follows:

7433

$$7434 \quad C = A \otimes B = \begin{bmatrix} A_{0,0} \otimes B & A_{0,1} \otimes B & \dots & A_{0,n_A-1} \otimes B \\ A_{1,0} \otimes B & A_{1,1} \otimes B & \dots & A_{1,n_A-1} \otimes B \\ \vdots & \vdots & \ddots & \vdots \\ A_{m_A-1,0} \otimes B & A_{m_A-1,1} \otimes B & \dots & A_{m_A-1,n_A-1} \otimes B \end{bmatrix}$$

7435 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
7436 Kronecker product are defined as

$$7437 \quad C(i_A m_B + i_B, j_A n_B + j_B) = A_{i_A, j_A} \otimes B_{i_B, j_B},$$

7438 where \otimes is the multiplicative operator specified by the `op` parameter.

7439 Logically, this operation occurs in three steps:

7440 **Setup** The internal matrices and mask used in the computation are formed and their domains
7441 and dimensions are tested for compatibility.

7442 **Compute** The indicated computations are carried out.

7443 **Output** The result is written into the output matrix, possibly under control of a mask.

7444 Up to four argument matrices are used in the `GrB_kronecker` operation:

- 7445 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7446 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)
- 7447 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 7448 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

7449 The argument matrices, the "product" operator (`op`), and the accumulation operator (if provided)
7450 are tested for domain compatibility as follows:

- 7451 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
7452 must be from one of the pre-defined types of Table 3.2.
- 7453 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 7454 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 7455 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 7456 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
7457 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
7458 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.

7492 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}) \times \mathbf{nrows}(\tilde{\mathbf{B}}), \mathbf{ncols}(\tilde{\mathbf{A}}) \times \mathbf{ncols}(\tilde{\mathbf{B}}), \{(i, j, T_{ij}) \text{ where } i =$
7493 $i_A \cdot m_B + i_B, j = j_A \cdot n_B + j_B, \forall (i_A, j_A) = \mathbf{ind}(\tilde{\mathbf{A}}), (i_B, j_B) = \mathbf{ind}(\tilde{\mathbf{B}})\}$ is created. The value of
7494 each of its elements is computed by

$$7495 \quad T_{i_A \cdot m_B + i_B, j_A \cdot n_B + j_B} = \tilde{\mathbf{A}}(i_A, j_A) \otimes \tilde{\mathbf{B}}(i_B, j_B),$$

7496 where \otimes is the multiplicative operator specified by the `op` parameter.

7497 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 7498 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 7499 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$7500 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

7501 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
7502 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$7503 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$7504 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$7505 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

7506 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

7509 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
7510 using what is called a *standard matrix mask and replace*. This is carried out under control of the
7511 mask which acts as a “write mask”.

- 7512 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
7513 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$7514 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7515 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
7516 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
7517 mask are unchanged:

$$7518 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7519 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
7520 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
7521 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
7522 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
7523 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_get(GrB_Scalar,GrB_Scalar,GrB_Field)	GrB_Scalar_get_Scalar(GrB_Scalar,GrB_Scalar,GrB_Field)
GrB_get(GrB_Scalar,char*,GrB_Field)	GrB_Scalar_get_String(GrB_Scalar,char*,GrB_Field)
GrB_get(GrB_Scalar,int32_t*,GrB_Field)	GrB_Scalar_get_INT32(GrB_Scalar,int32_t*,GrB_Field)
GrB_get(GrB_Scalar,size_t*,GrB_Field)	GrB_Scalar_get_SIZE(GrB_Scalar,size_t*,GrB_Field)
GrB_get(GrB_Scalar,void*,GrB_Field)	GrB_Scalar_get_VOID(GrB_Scalar,void*,GrB_Field)
GrB_get(GrB_Vector,GrB_Scalar,GrB_Field)	GrB_Vector_get_Scalar(GrB_Vector,GrB_Scalar,GrB_Field)
GrB_get(GrB_Vector,char*,GrB_Field)	GrB_Vector_get_String(GrB_Vector,char*,GrB_Field)
GrB_get(GrB_Vector,int32_t*,GrB_Field)	GrB_Vector_get_INT32(GrB_Vector,int32_t*,GrB_Field)
GrB_get(GrB_Vector,size_t*,GrB_Field)	GrB_Vector_get_SIZE(GrB_Vector,size_t*,GrB_Field)
GrB_get(GrB_Vector,void*,GrB_Field)	GrB_Vector_get_VOID(GrB_Vector,void*,GrB_Field)
GrB_get(GrB_Matrix,GrB_Scalar,GrB_Field)	GrB_Matrix_get_Scalar(GrB_Matrix,GrB_Scalar,GrB_Field)
GrB_get(GrB_Matrix,char*,GrB_Field)	GrB_Matrix_get_String(GrB_Matrix,char*,GrB_Field)
GrB_get(GrB_Matrix,int32_t*,GrB_Field)	GrB_Matrix_get_INT32(GrB_Matrix,int32_t*,GrB_Field)
GrB_get(GrB_Matrix,size_t*,GrB_Field)	GrB_Matrix_get_SIZE(GrB_Matrix,size_t*,GrB_Field)
GrB_get(GrB_Matrix,void*,GrB_Field)	GrB_Matrix_get_VOID(GrB_Matrix,void*,GrB_Field)
GrB_get(GrB_UnaryOp,GrB_Scalar,GrB_Field)	GrB_UnaryOp_get_Scalar(GrB_UnaryOp,GrB_Scalar,GrB_Field)
GrB_get(GrB_UnaryOp,char*,GrB_Field)	GrB_UnaryOp_get_String(GrB_UnaryOp,char*,GrB_Field)
GrB_get(GrB_UnaryOp,int32_t*,GrB_Field)	GrB_UnaryOp_get_INT32(GrB_UnaryOp,int32_t*,GrB_Field)
GrB_get(GrB_UnaryOp,size_t*,GrB_Field)	GrB_UnaryOp_get_SIZE(GrB_UnaryOp,size_t*,GrB_Field)
GrB_get(GrB_UnaryOp,void*,GrB_Field)	GrB_UnaryOp_get_VOID(GrB_UnaryOp,void*,GrB_Field)
GrB_get(GrB_IndexUnaryOp,GrB_Scalar,GrB_Field)	GrB_IndexUnaryOp_get_Scalar(GrB_IndexUnaryOp,GrB_Scalar,GrB_Field)
GrB_get(GrB_IndexUnaryOp,char*,GrB_Field)	GrB_IndexUnaryOp_get_String(GrB_IndexUnaryOp,char*,GrB_Field)
GrB_get(GrB_IndexUnaryOp,int32_t*,GrB_Field)	GrB_IndexUnaryOp_get_INT32(GrB_IndexUnaryOp,int32_t*,GrB_Field)
GrB_get(GrB_IndexUnaryOp,size_t*,GrB_Field)	GrB_IndexUnaryOp_get_SIZE(GrB_IndexUnaryOp,size_t*,GrB_Field)
GrB_get(GrB_IndexUnaryOp,void*,GrB_Field)	GrB_IndexUnaryOp_get_VOID(GrB_IndexUnaryOp,void*,GrB_Field)
GrB_get(GrB_BinaryOp,GrB_Scalar,GrB_Field)	GrB_BinaryOp_get_Scalar(GrB_BinaryOp,GrB_Scalar,GrB_Field)
GrB_get(GrB_BinaryOp,char*,GrB_Field)	GrB_BinaryOp_get_String(GrB_BinaryOp,char*,GrB_Field)
GrB_get(GrB_BinaryOp,int32_t*,GrB_Field)	GrB_BinaryOp_get_INT32(GrB_BinaryOp,int32_t*,GrB_Field)
GrB_get(GrB_BinaryOp,size_t*,GrB_Field)	GrB_BinaryOp_get_SIZE(GrB_BinaryOp,size_t*,GrB_Field)
GrB_get(GrB_BinaryOp,void*,GrB_Field)	GrB_BinaryOp_get_VOID(GrB_BinaryOp,void*,GrB_Field)
GrB_get(GrB_Monoid,GrB_Scalar,GrB_Field)	GrB_Monoid_get_Scalar(GrB_Monoid,GrB_Scalar,GrB_Field)
GrB_get(GrB_Monoid,char*,GrB_Field)	GrB_Monoid_get_String(GrB_Monoid,char*,GrB_Field)
GrB_get(GrB_Monoid,int32_t*,GrB_Field)	GrB_Monoid_get_INT32(GrB_Monoid,int32_t*,GrB_Field)
GrB_get(GrB_Monoid,size_t*,GrB_Field)	GrB_Monoid_get_SIZE(GrB_Monoid,size_t*,GrB_Field)
GrB_get(GrB_Monoid,void*,GrB_Field)	GrB_Monoid_get_VOID(GrB_Monoid,void*,GrB_Field)
GrB_get(GrB_Semiring,GrB_Scalar,GrB_Field)	GrB_Semiring_get_Scalar(GrB_Semiring,GrB_Scalar,GrB_Field)
GrB_get(GrB_Semiring,char*,GrB_Field)	GrB_Semiring_get_String(GrB_Semiring,char*,GrB_Field)
GrB_get(GrB_Semiring,int32_t*,GrB_Field)	GrB_Semiring_get_INT32(GrB_Semiring,int32_t*,GrB_Field)
GrB_get(GrB_Semiring,size_t*,GrB_Field)	GrB_Semiring_get_SIZE(GrB_Semiring,size_t*,GrB_Field)
GrB_get(GrB_Semiring,void*,GrB_Field)	GrB_Semiring_get_VOID(GrB_Semiring,void*,GrB_Field)
GrB_get(GrB_Descriptor,GrB_Scalar,GrB_Field)	GrB_Descriptor_get_Scalar(GrB_Descriptor,GrB_Scalar,GrB_Field)
GrB_get(GrB_Descriptor,char*,GrB_Field)	GrB_Descriptor_get_String(GrB_Descriptor,char*,GrB_Field)
GrB_get(GrB_Descriptor,int32_t*,GrB_Field)	GrB_Descriptor_get_INT32(GrB_Descriptor,int32_t*,GrB_Field)
GrB_get(GrB_Descriptor,size_t*,GrB_Field)	GrB_Descriptor_get_SIZE(GrB_Descriptor,size_t*,GrB_Field)
GrB_get(GrB_Descriptor,void*,GrB_Field)	GrB_Descriptor_get_VOID(GrB_Descriptor,void*,GrB_Field)
GrB_get(GrB_Type,GrB_Scalar,GrB_Field)	GrB_Type_get_Scalar(GrB_Type,GrB_Scalar,GrB_Field)
GrB_get(GrB_Type,char*,GrB_Field)	GrB_Type_get_String(GrB_Type,char*,GrB_Field)
GrB_get(GrB_Type,int32_t*,GrB_Field)	GrB_Type_get_INT32(GrB_Type,int32_t*,GrB_Field)
GrB_get(GrB_Type,size_t*,GrB_Field)	GrB_Type_get_SIZE(GrB_Type,size_t*,GrB_Field)
GrB_get(GrB_Type,void*,GrB_Field)	GrB_Type_get_VOID(GrB_Type,void*,GrB_Field)
GrB_get(GrB_Global,GrB_Scalar,GrB_Field)	GrB_Global_get_Scalar(GrB_Global,GrB_Scalar,GrB_Field)
GrB_get(GrB_Global,char*,GrB_Field)	GrB_Global_get_String(GrB_Global,char*,GrB_Field)
GrB_get(GrB_Global,int32_t*,GrB_Field)	GrB_Global_get_INT32(GrB_Global,int32_t*,GrB_Field)
GrB_get(GrB_Global,size_t*,GrB_Field)	GrB_Global_get_SIZE(GrB_Global,size_t*,GrB_Field)
GrB_get(GrB_Global,void*,GrB_Field)	GrB_Global_get_VOID(GrB_Global,void*,GrB_Field)

Table 5.13: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_set(GrB_Scalar,GrB_Scalar,GrB_Field)	GrB_Scalar_set_Scalar(GrB_Scalar,GrB_Scalar,GrB_Field)
GrB_set(GrB_Scalar,char*,GrB_Field)	GrB_Scalar_set_String(GrB_Scalar,char*,GrB_Field)
GrB_set(GrB_Scalar,int32_t,GrB_Field)	GrB_Scalar_set_INT32(GrB_Scalar,int32_t,GrB_Field)
GrB_set(GrB_Scalar,void*,GrB_Field,size_t)	GrB_Scalar_set_VOID(GrB_Scalar,void*,GrB_Field,size_t)
GrB_set(GrB_Vector,GrB_Scalar,GrB_Field)	GrB_Vector_set_Scalar(GrB_Vector,GrB_Scalar,GrB_Field)
GrB_set(GrB_Vector,char*,GrB_Field)	GrB_Vector_set_String(GrB_Vector,char*,GrB_Field)
GrB_set(GrB_Vector,int32_t,GrB_Field)	GrB_Vector_set_INT32(GrB_Vector,int32_t,GrB_Field)
GrB_set(GrB_Vector,void*,GrB_Field,size_t)	GrB_Vector_set_VOID(GrB_Vector,void*,GrB_Field,size_t)
GrB_set(GrB_Matrix,GrB_Scalar,GrB_Field)	GrB_Matrix_set_Scalar(GrB_Matrix,GrB_Scalar,GrB_Field)
GrB_set(GrB_Matrix,char*,GrB_Field)	GrB_Matrix_set_String(GrB_Matrix,char*,GrB_Field)
GrB_set(GrB_Matrix,int32_t,GrB_Field)	GrB_Matrix_set_INT32(GrB_Matrix,int32_t,GrB_Field)
GrB_set(GrB_Matrix,void*,GrB_Field,size_t)	GrB_Matrix_set_VOID(GrB_Matrix,void*,GrB_Field,size_t)
GrB_set(GrB_UnaryOp,GrB_Scalar,GrB_Field)	GrB_UnaryOp_set_Scalar(GrB_UnaryOp,GrB_Scalar,GrB_Field)
GrB_set(GrB_UnaryOp,char*,GrB_Field)	GrB_UnaryOp_set_String(GrB_UnaryOp,char*,GrB_Field)
GrB_set(GrB_UnaryOp,int32_t,GrB_Field)	GrB_UnaryOp_set_INT32(GrB_UnaryOp,int32_t,GrB_Field)
GrB_set(GrB_UnaryOp,void*,GrB_Field,size_t)	GrB_UnaryOp_set_VOID(GrB_UnaryOp,void*,GrB_Field,size_t)
GrB_set(GrB_IndexUnaryOp,GrB_Scalar,GrB_Field)	GrB_IndexUnaryOp_set_Scalar(GrB_IndexUnaryOp,GrB_Scalar,GrB_Field)
GrB_set(GrB_IndexUnaryOp,char*,GrB_Field)	GrB_IndexUnaryOp_set_String(GrB_IndexUnaryOp,char*,GrB_Field)
GrB_set(GrB_IndexUnaryOp,int32_t,GrB_Field)	GrB_IndexUnaryOp_set_INT32(GrB_IndexUnaryOp,int32_t,GrB_Field)
GrB_set(GrB_IndexUnaryOp,void*,GrB_Field,size_t)	GrB_IndexUnaryOp_set_VOID(GrB_IndexUnaryOp,void*,GrB_Field,size_t)
GrB_set(GrB_BinaryOp,GrB_Scalar,GrB_Field)	GrB_BinaryOp_set_Scalar(GrB_BinaryOp,GrB_Scalar,GrB_Field)
GrB_set(GrB_BinaryOp,char*,GrB_Field)	GrB_BinaryOp_set_String(GrB_BinaryOp,char*,GrB_Field)
GrB_set(GrB_BinaryOp,int32_t,GrB_Field)	GrB_BinaryOp_set_INT32(GrB_BinaryOp,int32_t,GrB_Field)
GrB_set(GrB_BinaryOp,void*,GrB_Field,size_t)	GrB_BinaryOp_set_VOID(GrB_BinaryOp,void*,GrB_Field,size_t)
GrB_set(GrB_Monoid,GrB_Scalar,GrB_Field)	GrB_Monoid_set_Scalar(GrB_Monoid,GrB_Scalar,GrB_Field)
GrB_set(GrB_Monoid,char*,GrB_Field)	GrB_Monoid_set_String(GrB_Monoid,char*,GrB_Field)
GrB_set(GrB_Monoid,int32_t,GrB_Field)	GrB_Monoid_set_INT32(GrB_Monoid,int32_t,GrB_Field)
GrB_set(GrB_Monoid,void*,GrB_Field,size_t)	GrB_Monoid_set_VOID(GrB_Monoid,void*,GrB_Field,size_t)
GrB_set(GrB_Semiring,GrB_Scalar,GrB_Field)	GrB_Semiring_set_Scalar(GrB_Semiring,GrB_Scalar,GrB_Field)
GrB_set(GrB_Semiring,char*,GrB_Field)	GrB_Semiring_set_String(GrB_Semiring,char*,GrB_Field)
GrB_set(GrB_Semiring,int32_t,GrB_Field)	GrB_Semiring_set_INT32(GrB_Semiring,int32_t,GrB_Field)
GrB_set(GrB_Semiring,void*,GrB_Field,size_t)	GrB_Semiring_set_VOID(GrB_Semiring,void*,GrB_Field,size_t)
GrB_set(GrB_Descriptor,GrB_Scalar,GrB_Field)	GrB_Descriptor_set_Scalar(GrB_Descriptor,GrB_Scalar,GrB_Field)
GrB_set(GrB_Descriptor,char*,GrB_Field)	GrB_Descriptor_set_String(GrB_Descriptor,char*,GrB_Field)
GrB_set(GrB_Descriptor,int32_t,GrB_Field)	GrB_Descriptor_set_INT32(GrB_Descriptor,int32_t,GrB_Field)
GrB_set(GrB_Descriptor,void*,GrB_Field,size_t)	GrB_Descriptor_set_VOID(GrB_Descriptor,void*,GrB_Field,size_t)
GrB_set(GrB_Type,GrB_Scalar,GrB_Field)	GrB_Type_set_Scalar(GrB_Type,GrB_Scalar,GrB_Field)
GrB_set(GrB_Type,char*,GrB_Field)	GrB_Type_set_String(GrB_Type,char*,GrB_Field)
GrB_set(GrB_Type,int32_t,GrB_Field)	GrB_Type_set_INT32(GrB_Type,int32_t,GrB_Field)
GrB_set(GrB_Type,void*,GrB_Field,size_t)	GrB_Type_set_VOID(GrB_Type,void*,GrB_Field,size_t)
GrB_set(GrB_Global,GrB_Scalar,GrB_Field)	GrB_Global_set_Scalar(GrB_Global,GrB_Scalar,GrB_Field)
GrB_set(GrB_Global,char*,GrB_Field)	GrB_Global_set_String(GrB_Global,char*,GrB_Field)
GrB_set(GrB_Global,int32_t,GrB_Field)	GrB_Global_set_INT32(GrB_Global,int32_t,GrB_Field)
GrB_set(GrB_Global,void*,GrB_Field,size_t)	GrB_Global_set_VOID(GrB_Global,void*,GrB_Field,size_t)

Appendix A

Revision history

Changes in 2.0.1 (Released: ## Xxxxx 2022:

- (Issue GH-69) Fix error in description of contents of matrix constructed from `GrB_Matrix_diag`.

Changes in 2.0.0 (Released: 15 November 2021:

- Reorganized Chapters 2 and 3: Chapter 2 contains prose regarding the basic concepts captured in the API; Chapter 3 presents all of the enumerations, literals, data types, and predefined objects required by the API. Made short captions for the List of Tables.
- (Issue BB-49, BB-50) Updated and corrected language regarding multithreading and completion, and requirements regarding acquire-release memory orders. Methods that used to force complete no longer do.
- (Issue BB-74, BB-9) Assigned integer values to all return codes as well as all enumerations in the API to ensure run-time compatibility between libraries.
- (Issues BB-70, BB-67) Changed semantics and signature of `GrB_wait(obj, mode)`. Added wait modes for 'complete' or 'materialize' and removed `GrB_wait(void)`. **This breaks backward compatibility.**
- (Issue GH-51) Removed deprecated `GrB_SCMP` literal from descriptor values. **This breaks backward compatibility.**
- (Issues BB-8, BB-36) Added sparse `GrB_Scalar` object and its use in additional variants of `extract/setElement` methods, and `reduce`, `apply`, `assign` and `select` operations.
- (Issues BB-34, GH-33, GH-45) Added new `select` operation that uses an index unary operator. Added new variants of `apply` that take an index unary operator (matrix and vector variants).
- (Issues BB-68, BB-51) Added `serialize` and `deserialize` methods for matrices to/from implementation defined formats.

- 7554 • (Issues BB-25, GH-42) Added import and export methods for matrices to/from API specified
7555 formats. Three formats have been specified: CSC, CSR, COO. Dense row and column formats
7556 have been deferred.
- 7557 • (Issue BB-75) Added matrix constructor to build a diagonal `GrB_Matrix` from a `GrB_Vector`.
- 7558 • (Issue BB-73) Allow `GrB_NULL` for dup operator in matrix and vector `build` methods. Return
7559 error if duplicate locations encountered.
- 7560 • (Issue BB-58) Added matrix and vector methods to remove (annihilate) elements.
- 7561 • (Issue BB-17) Added `GrB_ABS_T` (absolute value) unary operator.
- 7562 • (Issue GH-46) Adding `GrB_ONEB_T` binary operator that returns 1 cast to type `T` (not to
7563 be confused with the proposed unary operator).
- 7564 • (Issue GH-53) Added language about what constitutes a “conformant” implementation. Added
7565 `GrB_NOT_IMPLEMENTED` return value (API error) for API any combinations of inputs to
7566 a method that is not supported by the implementation.
- 7567 • Added `GrB_EMPTY_OBJECT` return value (execution error) that is used when an opaque
7568 object (currently only `GrB_Scalar`) is passed as an input that cannot be empty.
- 7569 • (Issue BB-45) Removed language about annihilators.
- 7570 • (Issue BB-69) Made names/symbols containing underscores searchable in PDF.
- 7571 • Updated a number algorithms in the appendix to use new operations and methods.
- 7572 • Numerous additions (some changes) to the non-polymorphic interface to track changes to the
7573 specification.
- 7574 • Typographical error in version macros was corrected. They are all caps: `GRB_VERSION` and
7575 `GRB_SUBVERSION`.
- 7576 • Typographical change to `eWiseAdd` Description to be consistent in order of set intersections.
- 7577 • Typographical errors in `eWiseAdd`: cut-and-paste errors from `eWiseMult`/set intersection
7578 fixed to read `eWiseAdd`/set union.
- 7579 • Typographical error (`NEQ` \rightarrow `NE`) in Description of Table 3.8.

7580 Changes in 1.3.0 (Released: 25 September 2019):

- 7581 • (Issue BB-50) Changed definition of completion and added `GrB_wait()` that takes an opaque
7582 GraphBLAS object as an argument.
- 7583 • (Issue BB-39) Added `GrB_kronecker` operation.
- 7584 • (Issue BB-40) Added variants of the `GrB_apply` operation that take a binary function and a
7585 scalar.

7586
7587

7588

7589
7590

7591
7592

7593

7594
7595

7596
7597

7598
7599

7600

7601
7602

7603

7604
7605

7606
7607

7608

7609
7610

7611

7612
7613

7614
7615

7616

7617

7618

- (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.

- 7619 • Fixed miscellaneous typographical errors (such as \otimes , \oplus).

7620 Changes in 1.2.0:

- 7621 • Removed "provisional" clause.

7622 Changes in 1.1.0:

- 7623 • Removed unnecessary `const` from `nindices`, `nrows`, and `ncols` parameters of both `extract` and
7624 `assign` operations.
- 7625 • Signature of `GrB_UnaryOp_new` changed: order of input parameters changed.
- 7626 • Signature of `GrB_BinaryOp_new` changed: order of input parameters changed.
- 7627 • Signature of `GrB_Monoid_new` changed: removal of domain argument which is now inferred
7628 from the domains of the binary operator provided.
- 7629 • Signature of `GrB_Vector_extractTuples` and `GrB_Matrix_extractTuples` to add an in/out ar-
7630 gument, `n`, which indicates the size of the output arrays provided (in terms of number of
7631 elements, not number of bytes). Added new execution error, `GrB_INSUFFICIENT_SPACE`
7632 which is returned when the capacities of the output arrays are insufficient to hold all of the
7633 tuples.
- 7634 • Changed `GrB_Column_assign` to `GrB_Col_assign` for consistency in non-polymorphic inter-
7635 face.
- 7636 • Added replace flag (`z`) notation to Table 4.1.
- 7637 • Updated the "Mathematical Description" of the `assign` operation in Table 4.1.
- 7638 • Added triangle counting example.
- 7639 • Added subsection headers for `accumulate` and `mask/replace` discussions in the Description
7640 sections of GraphBLAS operations when the respective text was the "standard" text (i.e.,
7641 identical in a majority of the operations).
- 7642 • Fixed typographical errors.

7643 Changes in 1.0.2:

- 7644 • Expanded the definitions of `Vector_build` and `Matrix_build` to conceptually use intermediate
7645 matrices and avoid casting issues in certain implementations.
- 7646 • Fixed the bug in the `GrB_assign` definition. Elements of the output object are no longer being
7647 erased outside the assigned area.
- 7648 • Changes non-polymorphic interface:
 - 7649 – Renamed `GrB_Row_extract` to `GrB_Col_extract`.

- 7650 – Renamed GrB_Vector_reduce_BinaryOp to GrB_Matrix_reduce_BinaryOp.
- 7651 – Renamed GrB_Vector_reduce_Monoid to GrB_Matrix_reduce_Monoid.
- 7652 • Fixed the bugs with respect to isolated vertices in the Maximal Independent Set example.
- 7653 • Fixed numerous typographical errors.

Appendix B

Non-opaque data format definitions

B.1 GrB_Format: Specify the format for input/output of a Graph-BLAS 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.

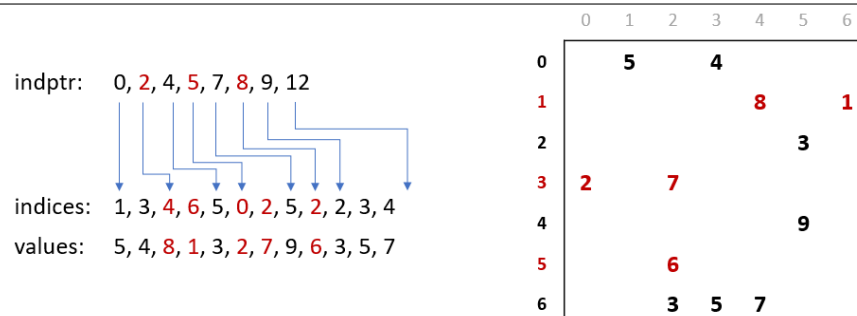


Figure B.1: Data layout for CSR format.

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.

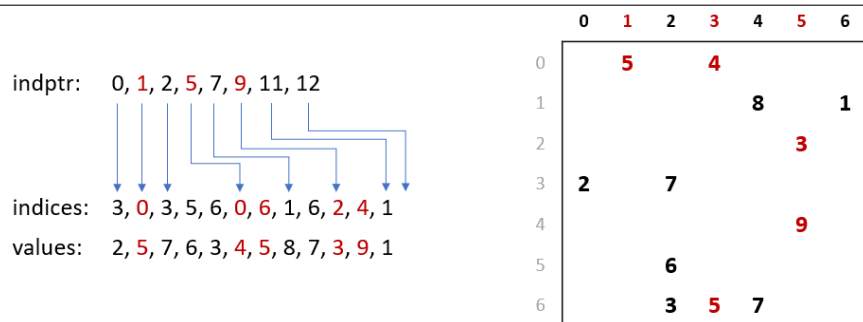


Figure B.2: Data layout for CSC format.

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.

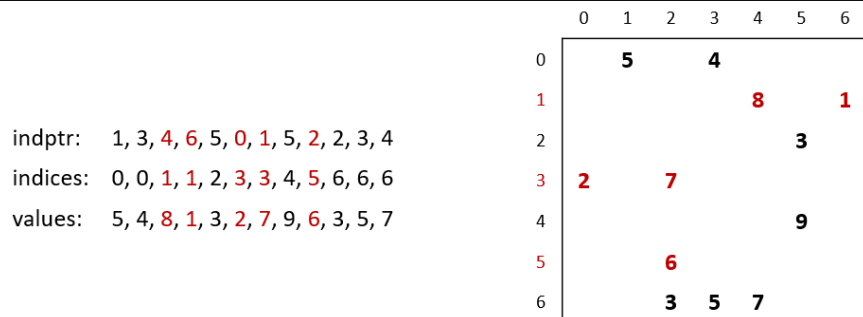


Figure B.3: Data layout for COO format.

7686 **Appendix C**

7687 **Examples**

C.1 Example: Level breadth-first search (BFS) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS traversal
9   * of the graph and sets  $v[i]$  to the level in which vertex  $i$  is visited ( $v[s] == 1$ ).
10  * If  $i$  is not reachable from  $s$ , then  $v[i] = 0$ . (Vector  $v$  should be empty on input.)
11  */
12  GrB_Info BFS(GrB_Vector *v, GrB_Matrix A, GrB_Index s)
13  {
14      GrB_Index n;
15      GrB_Matrix_nrows(&n,A);                //  $n = \#$  of rows of  $A$ 
16
17      GrB_Vector_new(v,GrB_INT32,n);          // Vector<int32_t>  $v(n)$ 
18
19      GrB_Vector q;                          // vertices visited in each level
20      GrB_Vector_new(&q,GrB_BOOL,n);          // Vector<bool>  $q(n)$ 
21      GrB_Vector_setElement(q,(bool)true,s);  //  $q[s] = \text{true}$ , false everywhere else
22
23      /*
24       * BFS traversal and label the vertices.
25       */
26      int32_t d = 0;                          //  $d = \text{level in BFS traversal}$ 
27      bool succ = false;                      //  $\text{succ} == \text{true}$  when some successor found
28      do {
29          ++d;                                // next level (start with 1)
30          GrB_assign(*v,q,GrB_NULL,d,GrB_ALL,n,GrB_NULL); //  $v[q] = d$ 
31          GrB_vxm(q,*v,GrB_NULL,GrB_LOR_LAND_SEMIRING_BOOL,
32                q,A,GrB_DESC_RC);             //  $q[!v] = q \parallel A$ ; finds all the
33                                              // unvisited successors from current  $q$ 
34          GrB_reduce(&succ,GrB_NULL,GrB_LOR_MONOID_BOOL,
35                q,GrB_NULL);                  //  $\text{succ} = \parallel(q)$ 
36      } while (succ);                          // if there is no successor in  $q$ , we are done.
37
38      GrB_free(&q);                            //  $q$  vector no longer needed
39
40      return GrB_SUCCESS;
41  }

```

C.2 Example: Level BFS in GraphBLAS using apply

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS traversal
9   * of the graph and sets  $v[i]$  to the level in which vertex  $i$  is visited ( $v[s] == 1$ ).
10  * If  $i$  is not reachable from  $s$ , then  $v[i]$  does not have a stored element.
11  * Vector  $v$  should be uninitialized on input.
12  */
13  GrB_Info BFS(GrB_Vector *v, const GrB_Matrix A, GrB_Index s)
14  {
15      GrB_Index n;
16      GrB_Matrix_nrows(&n,A);           //  $n = \#$  of rows of  $A$ 
17
18      GrB_Vector_new(v,GrB_INT32,n);     // Vector<int32_t>  $v(n) = 0$ 
19
20      GrB_Vector q;                     // vertices visited in each level
21      GrB_Vector_new(&q,GrB_BOOL,n);    // Vector<bool>  $q(n) = \text{false}$ 
22      GrB_Vector_setElement(q,(bool)true,s); //  $q[s] = \text{true}$ , false everywhere else
23
24      /*
25       * BFS traversal and label the vertices.
26       */
27      int32_t level = 0;                // level = depth in BFS traversal
28      GrB_Index nvals;
29      do {
30          ++level;                     // next level (start with 1)
31          GrB_apply(*v,GrB_NULL,GrB_PLUS_INT32,
32                  GrB_SECOND_INT32,q,level,GrB_NULL); //  $v[q] = \text{level}$ 
33          GrB_vxm(q,*v,GrB_NULL,GrB_LOR_LAND_SEMIRING_BOOL,
34                  q,A,GrB_DESC_RC);    //  $q[!v] = q \ || \ \&\& \ A$ ; finds all the
35                                      // unvisited successors from current  $q$ 
36          GrB_Vector_nvals(&nvals, q);
37      } while (nvals);                // if there is no successor in  $q$ , we are done.
38
39      GrB_free(&q);                   //  $q$  vector no longer needed
40
41      return GrB_SUCCESS;
42  }

```

C.3 Example: Parent BFS in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a binary  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS
9   * traversal of the graph and sets  $parents[i]$  to the index of vertex  $i$ 's parent.
10  * The parent of the root vertex,  $s$ , will be set to itself ( $parents[s] = s$ ). If
11  * vertex  $i$  is not reachable from  $s$ ,  $parents[i]$  will not contain a stored value.
12  */
13  GrB_Info BFS(GrB_Vector *parents, const GrB_Matrix A, GrB_Index s)
14  {
15      GrB_Index N;
16      GrB_Matrix_nrows(&N, A);           //  $N = \#$  vertices
17
18      GrB_Vector_new(parents, GrB_UINT64, N);
19      GrB_Vector_setElement(*parents, s, s);           //  $parents[s] = s$ 
20
21      GrB_Vector wavefront;
22      GrB_Vector_new(&wavefront, GrB_UINT64, N);
23      GrB_Vector_setElement(wavefront, 1UL, s);       //  $wavefront[s] = 1$ 
24
25      /*
26       * BFS traversal and label the vertices.
27       */
28      GrB_Index nvals;
29      GrB_Vector_nvals(&nvals, wavefront);
30
31      while (nvals > 0)
32      {
33          // convert all stored values in wavefront to their 0-based index
34          GrB_apply(wavefront, GrB_NULL, GrB_NULL, GrB_ROWINDEX_INT64,
35                  wavefront, 0UL, GrB_NULL);
36
37          // "FIRST" because left-multiplying wavefront rows. Masking out the parent
38          // list ensures wavefront values do not overwrite parents already stored.
39          GrB_vxm(wavefront, *parents, GrB_NULL, GrB_MIN_FIRST_SEMIRING_UINT64,
40                  wavefront, A, GrB_DESC_RSC);
41
42          // Don't need to mask here since we did it in vxm. Merges new parents in
43          // current wavefront with existing parents:  $parents += wavefront$ 
44          GrB_apply(*parents, GrB_NULL, GrB_PLUS_UINT64,
45                  GrB_IDENTITY_UINT64, wavefront, GrB_NULL);
46
47          GrB_Vector_nvals(&nvals, wavefront);
48      }
49
50      GrB_free(&wavefront);
51
52      return GrB_SUCCESS;
53  }

```


C.4 Example: Betweenness centrality (BC) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ ,
9   * compute the BC-metric vector  $\delta$ , which should be empty on input.
10  */
11 GrB_Info BC(GrB_Vector *delta, GrB_Matrix A, GrB_Index s)
12 {
13     GrB_Index n;
14     GrB_Matrix_nrows(&n, A);                //  $n = \#$  of vertices in graph
15
16     GrB_Vector_new(delta, GrB_FP32, n);      // Vector<float>  $\delta(n)$ 
17
18     GrB_Matrix sigma;
19     GrB_Matrix_new(&sigma, GrB_INT32, n, n); //  $\text{Matrix}<\text{int32}_t> \text{sigma}(n, n)$ 
20                                           //  $\text{sigma}[d, k] = \#$  shortest paths to node  $k$  at level  $d$ 
21
22     GrB_Vector q;
23     GrB_Vector_new(&q, GrB_INT32, n);        // Vector<int32_t>  $q(n)$  of path counts
24     GrB_Vector_setElement(q, 1, s);          //  $q[s] = 1$ 
25
26     GrB_Vector p;
27     GrB_Vector_dup(&p, q);                   // Vector<int32_t>  $p(n)$  shortest path counts so far
28                                           //  $p = q$ 
29
30     GrB_vxm(q, p, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
31             q, A, GrB_DESC_RC);              // get the first set of out neighbors
32
33     /*
34     * BFS phase
35     */
36     GrB_Index d = 0;                          // BFS level number
37     int32_t sum = 0;                          //  $\text{sum} == 0$  when BFS phase is complete
38
39     do {
40         GrB_assign(sigma, GrB_NULL, GrB_NULL, q, d, GrB_ALL, n, GrB_NULL); //  $\text{sigma}[d, :] = q$ 
41         GrB_eWiseAdd(p, GrB_NULL, GrB_NULL, GrB_PLUS_INT32, p, q, GrB_NULL); // accum path counts on this level
42         GrB_vxm(q, p, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
43                 q, A, GrB_DESC_RC);          //  $q = \#$  paths to nodes reachable
44                                           // from current level
45         GrB_reduce(&sum, GrB_NULL, GrB_PLUS_MONOID_INT32, q, GrB_NULL); // sum path counts at this level
46         ++d;
47     } while (sum);
48
49     /*
50     * BC computation phase
51     * ( $t_1, t_2, t_3, t_4$ ) are temporary vectors
52     */
53     GrB_Vector t1; GrB_Vector_new(&t1, GrB_FP32, n);
54     GrB_Vector t2; GrB_Vector_new(&t2, GrB_FP32, n);
55     GrB_Vector t3; GrB_Vector_new(&t3, GrB_FP32, n);
56     GrB_Vector t4; GrB_Vector_new(&t4, GrB_FP32, n);
57
58     for (int i=d-1; i>0; i--)
59     {
60         GrB_assign(t1, GrB_NULL, GrB_NULL, 1.0f, GrB_ALL, n, GrB_NULL); //  $t_1 = 1 + \delta$ 
61         GrB_eWiseAdd(t1, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, t1, *delta, GrB_NULL);
62         GrB_extract(t2, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i, GrB_DESC_T0); //  $t_2 = \text{sigma}[i, :]$ 
63         GrB_eWiseMult(t2, GrB_NULL, GrB_NULL, GrB_DIV_FP32, t1, t2, GrB_NULL); //  $t_2 = (1 + \delta) / \text{sigma}[i, :]$ 
64         GrB_mvx(t3, GrB_NULL, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32,
65                 // add contributions made by

```

```

63         A, t2, GrB_NULL);
64     GrB_extract(t4, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i-1, GrB_DESC_T0); // t4 = sigma[i-1,:]
65     GrB_eWiseMult(t4, GrB_NULL, GrB_NULL, GrB_TIMES_FP32, t4, t3, GrB_NULL); // t4 = sigma[i-1,:]*t3
66     GrB_eWiseAdd(*delta, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, *delta, t4, GrB_NULL); // accumulate into delta
67 }
68
69 GrB_free(&sigma);
70 GrB_free(&q); GrB_free(&p);
71 GrB_free(&t1); GrB_free(&t2); GrB_free(&t3); GrB_free(&t4);
72
73 return GrB_SUCCESS;
74 }

```

C.5 Example: Batched BC in GraphBLAS

```

1  #include <stdlib.h>
2  #include "GraphBLAS.h" // in addition to other required C headers
3
4  // Compute partial BC metric for a subset of source vertices, s, in graph A
5  GrB_Info BC_update(GrB_Vector *delta, GrB_Matrix A, GrB_Index *s, GrB_Index nsver)
6  {
7      GrB_Index n;
8      GrB_Matrix_nrows(&n, A); // n = # of vertices in graph
9      GrB_Vector_new(delta, GrB_FP32, n); // Vector<float> delta(n)
10
11     // index and value arrays needed to build numsp
12     GrB_Index *i_nsver = (GrB_Index*) malloc(sizeof(GrB_Index)*nsver);
13     int32_t *ones = (int32_t*) malloc(sizeof(int32_t)*nsver);
14     for(int i=0; i<nsver; ++i) {
15         i_nsver[i] = i;
16         ones[i] = 1;
17     }
18
19     // numsp: structure holds the number of shortest paths for each node and starting vertex
20     // discovered so far. Initialized to source vertices: numsp[s[i],i]=1, i=[0,nsver)
21     GrB_Matrix numsp;
22     GrB_Matrix_new(&numsp, GrB_INT32, n, nsver);
23     GrB_Matrix_build(numsp, s, i_nsver, ones, nsver, GrB_PLUS_INT32);
24     free(i_nsver); free(ones);
25
26     // frontier: Holds the current frontier where values are path counts.
27     // Initialized to out vertices of each source node in s.
28     GrB_Matrix frontier;
29     GrB_Matrix_new(&frontier, GrB_INT32, n, nsver);
30     GrB_extract(frontier, numsp, GrB_NULL, A, GrB_ALL, n, s, nsver, GrB_DESC_RCT0);
31
32     // sigma: stores frontier information for each level of BFS phase. The memory
33     // for an entry in sigmas is only allocated within the do-while loop if needed.
34     // n is an upper bound on diameter.
35     GrB_Matrix *sigmas = (GrB_Matrix*) malloc(sizeof(GrB_Matrix)*n);
36
37     int32_t d = 0; // BFS level number
38     GrB_Index nvals = 0; // nvals == 0 when BFS phase is complete
39
40     // ----- The BFS phase (forward sweep) -----
41     do {
42         // sigmas[d](:,s) = dth level frontier from source vertex s
43         GrB_Matrix_new(&(sigmas[d]), GrB_BOOL, n, nsver);
44
45         GrB_apply(sigmas[d], GrB_NULL, GrB_NULL,
46                 GrB_IDENTITY_BOOL, frontier, GrB_NULL); // sigmas[d](:,:) = (Boolean) frontier
47         GrB_eWiseAdd(numsp, GrB_NULL, GrB_NULL, GrB_PLUS_INT32,
48                     numsp, frontier, GrB_NULL); // numsp += frontier (accum path counts)
49         GrB_mxm(frontier, numsp, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
50                 A, frontier, GrB_DESC_RCT0); // f<!numsp> = A' +.* f (update frontier)
51         GrB_Matrix_nvals(&nvals, frontier); // number of nodes in frontier at this level
52         d++;
53     } while (nvals);
54
55     // nspinv: the inverse of the number of shortest paths for each node and starting vertex.
56     GrB_Matrix nspinv;
57     GrB_Matrix_new(&nspinv, GrB_FP32, n, nsver);
58     GrB_apply(nspinv, GrB_NULL, GrB_NULL,
59              GrB_MINV_FP32, numsp, GrB_NULL); // nspinv = 1./numsp
60
61     // bcu: BC updates for each vertex for each starting vertex in s
62     GrB_Matrix bcu;

```

```

63 GrB_Matrix_new(&bcu, GrB_FP32, n, nsver);
64 GrB_assign(bcu, GrB_NULL, GrB_NULL,
65           1.0f, GrB_ALL, n, GrB_ALL, nsver, GrB_NULL); // filled with 1 to avoid sparsity issues
66
67 GrB_Matrix w; // temporary workspace matrix
68 GrB_Matrix_new(&w, GrB_FP32, n, nsver);
69
70 // ----- Tally phase (backward sweep) -----
71 for (int i=d-1; i>0; i--) {
72     GrB_eWiseMult(w, sigmas[i], GrB_NULL,
73                 GrB_TIMES_FP32, bcu, nspinv, GrB_DESC_R); // w<sigmas[i]>=(1 ./ nsp).*bcu
74
75     // add contributions by successors and mask with that BFS level's frontier
76     GrB_mxm(w, sigmas[i-1], GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32,
77            A, w, GrB_DESC_R); // w<sigmas[i-1]> = (A +.* w)
78     GrB_eWiseMult(bcu, GrB_NULL, GrB_PLUS_FP32, GrB_TIMES_FP32,
79                 w, numsp, GrB_NULL); // bcu += w .* numsp
80 }
81
82 // row reduce bcu and subtract "nsver" from every entry to account
83 // for 1 extra value per bcu row element.
84 GrB_reduce(*delta, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, bcu, GrB_NULL);
85 GrB_apply(*delta, GrB_NULL, GrB_NULL, GrB_MINUS_FP32, *delta, (float)nsver, GrB_NULL);
86
87 // Release resources
88 for (int i=0; i<d; i++) {
89     GrB_free(&(sigmas[i]));
90 }
91 free(sigmas);
92
93 GrB_free(&frontier); GrB_free(&numsp);
94 GrB_free(&nspinv); GrB_free(&bcu); GrB_free(&w);
95
96 return GrB_SUCCESS;
97 }

```

C.6 Example: Maximal independent set (MIS) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  // Assign a random number to each element scaled by the inverse of the node's degree.
8  // This will increase the probability that low degree nodes are selected and larger
9  // sets are selected.
10 void setRandom(void *out, const void *in)
11 {
12     uint32_t degree = *(uint32_t*)in;
13     *(float*)out = (0.0001f + random()/(1. + 2.*degree)); // add 1 to prevent divide by zero
14 }
15
16 /*
17  * A variant of Luby's randomized algorithm [Luby 1985].
18  *
19  * Given a numeric n x n adjacency matrix A of an unweighted and undirected graph (where
20  * the value true represents an edge), compute a maximal set of independent vertices and
21  * return it in a boolean n-vector, 'iset' where set[i] == true implies vertex i is a member
22  * of the set (the iset vector should be uninitialized on input.)
23  */
24 GrB_Info MIS(GrB_Vector *iset, const GrB_Matrix A)
25 {
26     GrB_Index n;
27     GrB_Matrix_nrows(&n,A); // n = # of rows of A
28
29     GrB_Vector prob; // holds random probabilities for each node
30     GrB_Vector neighbor_max; // holds value of max neighbor probability
31     GrB_Vector new_members; // holds set of new members to iset
32     GrB_Vector new_neighbors; // holds set of new neighbors to new iset mbrs.
33     GrB_Vector candidates; // candidate members to iset
34
35     GrB_Vector_new(&prob,GrB_FP32,n);
36     GrB_Vector_new(&neighbor_max,GrB_FP32,n);
37     GrB_Vector_new(&new_members,GrB_BOOL,n);
38     GrB_Vector_new(&new_neighbors,GrB_BOOL,n);
39     GrB_Vector_new(&candidates,GrB_BOOL,n);
40     GrB_Vector_new(iset,GrB_BOOL,n); // Initialize independent set vector, bool
41
42     GrB_UnaryOp set_random;
43     GrB_UnaryOp_new(&set_random,setRandom,GrB_FP32,GrB_UINT32);
44
45     // compute the degree of each vertex.
46     GrB_Vector degrees;
47     GrB_Vector_new(&degrees,GrB_FP64,n);
48     GrB_reduce(degrees,GrB_NULL,GrB_NULL,GrB_PLUS_FP64,A,GrB_NULL);
49
50     // Isolated vertices are not candidates: candidates[degrees != 0] = true
51     GrB_assign(candidates,degrees,GrB_NULL,true,GrB_ALL,n,GrB_NULL);
52
53     // add all singletons to iset: iset[degree == 0] = 1
54     GrB_assign(*iset,degrees,GrB_NULL,true,GrB_ALL,n,GrB_DESC_RC) ;
55
56     // Iterate while there are candidates to check.
57     GrB_Index nvals;
58     GrB_Vector_nvals(&nvals, candidates);
59     while (nvals > 0) {
60         // compute a random probability scaled by inverse of degree
61         GrB_apply(prob,candidates,GrB_NULL,set_random,degrees,GrB_DESC_R);
62     }

```

```

63 // compute the max probability of all neighbors
64 GrB_mnv(neighbor_max, candidates, GrB_NULL, GrB_MAX_SECOND_SEMIRING_FP32, A, prob, GrB_DESC_R);
65
66 // select vertex if its probability is larger than all its active neighbors,
67 // and apply a "masked no-op" to remove stored falses
68 GrB_eWiseAdd(new_members, GrB_NULL, GrB_NULL, GrB_GT_FP64, prob, neighbor_max, GrB_NULL);
69 GrB_apply(new_members, new_members, GrB_NULL, GrB_IDENTITY_BOOL, new_members, GrB_DESC_R);
70
71 // add new members to independent set.
72 GrB_eWiseAdd(*iset, GrB_NULL, GrB_NULL, GrB_LOR, *iset, new_members, GrB_NULL);
73
74 // remove new members from set of candidates  $c = c \ominus !new$ 
75 GrB_eWiseMult(candidates, new_members, GrB_NULL,
76               GrB_LAND, candidates, candidates, GrB_DESC_RC);
77
78 GrB_Vector_nvals(&nvals, candidates);
79 if (nvals == 0) { break; } // early exit condition
80
81 // Neighbors of new members can also be removed from candidates
82 GrB_mnv(new_neighbors, candidates, GrB_NULL, GrB_LOR_LAND_SEMIRING_BOOL,
83         A, new_members, GrB_NULL);
84 GrB_eWiseMult(candidates, new_neighbors, GrB_NULL, GrB_LAND,
85               candidates, candidates, GrB_DESC_RC);
86
87 GrB_Vector_nvals(&nvals, candidates);
88 }
89
90 GrB_free(&neighbor_max); // free all objects "new'ed"
91 GrB_free(&new_members);
92 GrB_free(&new_neighbors);
93 GrB_free(&prob);
94 GrB_free(&candidates);
95 GrB_free(&set_random);
96 GrB_free(&degrees);
97
98 return GrB_SUCCESS;
99 }

```

C.7 Example: Counting triangles in GraphBLAS

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <stdint.h>
4 #include <stdbool.h>
5 #include "GraphBLAS.h"
6
7 /*
8  * Given an  $n \times n$  boolean adjacency matrix,  $A$ , of an undirected graph, computes
9  * the number of triangles in the graph.
10 */
11 uint64_t triangle_count(GrB_Matrix A)
12 {
13     GrB_Index n;
14     GrB_Matrix_nrows(&n, A);           //  $n = \#$  of vertices
15
16     //  $L$ :  $N \times N$ , lower-triangular, bool
17     GrB_Matrix L;
18     GrB_Matrix_new(&L, GrB_BOOL, n, n);
19     GrB_select(L, GrB_NULL, GrB_NULL, GrB_TRIL, A, 0UL, GrB_NULL);
20
21     GrB_Matrix C;
22     GrB_Matrix_new(&C, GrB_UINT64, n, n);
23
24     GrB_mxm(C, L, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_UINT64, L, L, GrB_NULL); //  $C \langle L \rangle = L +.* L$ 
25
26     uint64_t count;
27     GrB_reduce(&count, GrB_NULL, GrB_PLUS_MONOID_UINT64, C, GrB_NULL); // 1-norm of  $C$ 
28
29     GrB_free(&C);
30     GrB_free(&L);
31
32     return count;
33 }
```