

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

Version 2.0.1

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

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

Generated on 2022/12/05 at 09:57:09 EDT

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

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

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

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

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

# Contents

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

48	3.4	Algebraic objects, operators and associated functions . . . . .	33
49	3.4.1	Operators . . . . .	34
50	3.4.2	Monoids . . . . .	39
51	3.4.3	Semirings . . . . .	39
52	3.5	Collections . . . . .	43
53	3.5.1	Scalars . . . . .	43
54	3.5.2	Vectors . . . . .	43
55	3.5.3	Matrices . . . . .	44
56	3.5.3.1	External matrix formats . . . . .	44
57	3.5.4	Masks . . . . .	44
58	3.6	Fields . . . . .	45
59	3.6.1	String Handling . . . . .	46
60	3.6.2	Hints . . . . .	46
61	3.6.3	get after set . . . . .	46
62	3.7	Descriptors . . . . .	48
63	3.8	GrB_Info return values . . . . .	48
64	<b>4</b>	<b>Methods</b>	<b>53</b>
65	4.1	Context methods . . . . .	53
66	4.1.1	init: Initialize a GraphBLAS context . . . . .	53
67	4.1.2	finalize: Finalize a GraphBLAS context . . . . .	54
68	4.1.3	getVersion: Get the version number of the standard. . . . .	55
69	4.2	Object methods . . . . .	55
70	4.2.1	Get and Set methods . . . . .	56
71	4.2.1.1	get: Query the value of an object . . . . .	56
72	4.2.1.2	set: Set field of an object . . . . .	57
73	4.2.2	Algebra methods . . . . .	58
74	4.2.2.1	Type_new: Construct a new GraphBLAS (user-defined) type . . . .	58
75	4.2.2.2	UnaryOp_new: Construct a new GraphBLAS unary operator . . . .	59
76	4.2.2.3	BinaryOp_new: Construct a new GraphBLAS binary operator . . .	60
77	4.2.2.4	Monoid_new: Construct a new GraphBLAS monoid . . . . .	62

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

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

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

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



# List of Tables

2.1	Types of GraphBLAS opaque objects. . . . .	23
2.2	Methods that forced completion prior to GraphBLAS v2.0. . . . .	28
3.1	Enumeration literals and corresponding values input to various GraphBLAS methods. . . . .	32
3.2	Predefined GrB_Type values. . . . .	33
3.3	Operator input for relevant GraphBLAS operations. . . . .	34
3.4	Properties and recipes for building GraphBLAS algebraic objects. . . . .	35
3.5	Predefined unary and binary operators for GraphBLAS in C. . . . .	37
3.6	Predefined index unary operators for GraphBLAS in C. . . . .	38
3.7	Predefined monoids for GraphBLAS in C. . . . .	40
3.8	Predefined “true” semirings for GraphBLAS in C. . . . .	41
3.9	Other useful predefined semirings for GraphBLAS in C. . . . .	42
3.10	GrB_Format enumeration literals and corresponding values for matrix import and export methods. . . . .	44
3.11	Field values of type GrB_Field enumeration, corresponding types, and the objects which must implement that GrB_Field. Collection refers to GrB_Matrix, GrB_Vector, and GrB_Scalar, Algebraic refers to Operators, Monoids, and Semirings, while All refers to all GraphBLAS objects. Global fields are denoted by Global. All fields may be read, some may be written (denoted by W), and some are hints (denoted by H) which may be ignored by the implementation. For * see 3.6.3 . . . . .	47
3.12	Descriptor types and literals for fields and values. . . . .	49
3.13	Predefined GraphBLAS descriptors. . . . .	50
3.14	Enumeration literals and corresponding values returned by GraphBLAS methods and operations. . . . .	51
4.1	A mathematical notation for the fundamental GraphBLAS operations supported in this specification. . . . .	119

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

# 222 List of Figures

223	3.1 Hierarchy of algebraic object classes in GraphBLAS. . . . .	43
224	4.1 Flowchart for the GraphBLAS operations. . . . .	120
225	B.1 Data layout for CSR format. . . . .	287
226	B.2 Data layout for CSC format. . . . .	288
227	B.3 Data layout for COO format. . . . .	288

## Acknowledgments

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

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

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

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

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

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

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

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

# Chapter 1

## Introduction

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

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

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

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

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

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

is not possible to express the signature of that method.

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

The remainder of this document is organized as follows:

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

## Chapter 2

# Basic concepts

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

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

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

## 2.1 Glossary

### 2.1.1 GraphBLAS API basic definitions

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

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

### 2.1.2 GraphBLAS objects and their structure

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



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

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

### 2.1.3 Algebraic structures used in the GraphBLAS

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

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

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

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

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

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

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

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

### 2.1.5 GraphBLAS methods: behaviors and error conditions

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

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

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

## 2.2 Notation

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

## 2.3 Mathematical foundations

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

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

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

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

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

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

---

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

Table 2.1: Types of GraphBLAS opaque objects.

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

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

## 2.4 GraphBLAS opaque objects

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

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

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

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

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

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

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

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

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

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

## 2.5 Execution model

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



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

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

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

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

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

### 2.5.1 Execution modes

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

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

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

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

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

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

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

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

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

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

## 2.5.2 Multi-threaded execution

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

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

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

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

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

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

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

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

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

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

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

## 2.6 Error model

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

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

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

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

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

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

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

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

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

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

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

### 793 3.3 Types (domains)

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

---

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

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

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

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

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

---



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

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

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

### 3.4 Algebraic objects, operators and associated functions

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

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

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

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

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

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

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

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

### 3.4.1 Operators

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

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

---

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

(a) Properties of algebraic objects.

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

(b) Recipes for algebraic objects.

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

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

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

---

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

842 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  
843 domains,  $D_{out}$ ,  $D_{in_1}$ ,  $D_{in_2}$ , the domain of GraphBLAS indices, and an operation  $f_i : D_{in_1} \times I_{U64}^2 \times$   
844  $D_{in_2} \rightarrow D_{out}$  (where  $I_{U64}$  corresponds to the domain of a `GrB_Index`). For a given GraphBLAS  
845 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  $\mathbf{f}(F_i) = f_i$ .

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

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

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

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

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

### 3.4.2 Monoids

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

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

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

### 3.4.3 Semirings

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

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

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

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

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

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

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

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

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



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

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

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

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

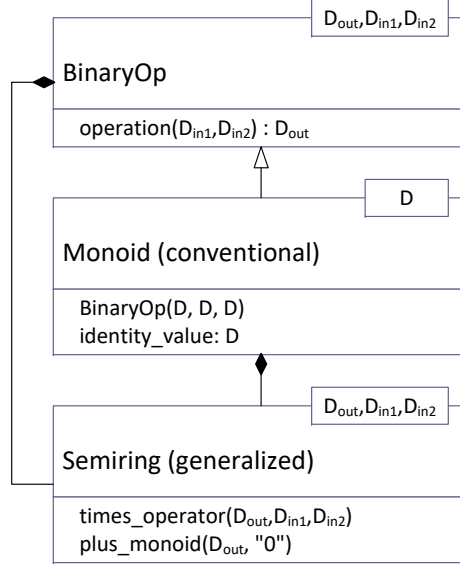


Figure 3.1: Hierarchy of algebraic object classes in GraphBLAS. GraphBLAS semirings consist of a conventional monoid with one domain for the addition function, and a binary operator with three domains for the multiplication function.

## 3.5 Collections

### 3.5.1 Scalars

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

### 3.5.2 Vectors

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

### 3.5.3 Matrices

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

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

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

#### 3.5.3.1 External matrix formats

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

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

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

### 3.5.4 Masks

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

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

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

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

A one-dimensional mask  $\mathbf{m} = \langle N, \{i\} \rangle$  is defined by its number of elements  $N > 0$ , and a set **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 **size**( $\mathbf{m}$ ) =  $N$ . The set **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 **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 **ncols**( $\mathbf{M}$ ) =  $N$ , and **nrows**( $\mathbf{M}$ ) =  $M$ . We also define the sets **indrow**( $\mathbf{M}$ ) =  $\{i : \exists (i, j) \in \mathbf{ind}(\mathbf{M})\}$  and **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 **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 **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 **ind**( $\neg \mathbf{M}$ ) =  $\{(i, j) : 0 \leq i < M, 0 \leq j < N, (i, j) \notin \mathbf{ind}(\mathbf{M})\}$ . It is the set of all possible indices that do not appear in  $\mathbf{M}$ .

## 3.6 Fields

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

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

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

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

968 by other operations, such as `GrB_BLOCKING_MODE` which is determined by `GrB_Init`.

969 The `GrB_NAME` field is a special case regarding writability. All objects which have a `GrB_NAME`  
970 field default to an empty string, `GrB_NAMESIZE` will be 0. Collections and `GrB_Descriptors` may  
971 have their `GrB_NAME` set at any time. Algebraic objects and `GrB_Types` may only have their  
972 `GrB_NAME` set once to a globally unique value. Attempting to set this field after it has already  
973 been set will return a `GrB_OUTPUT_NOT_EMPTY` error code.

### 974 3.6.1 String Handling

975 When the input to `GrB_<OBJ>_set` is a `char*` the array must be null terminated. The GraphBLAS  
976 implementation must copy this array into internal data structures.

977 When a `char*` is the output argument of `GrB_<OBJ>_get` the user must preallocate a properly sized  
978 buffer. The returned string must be null terminated, so the buffer must be at least 1 larger than  
979 the size of the string. For instance the buffer for `GrB_NAME` must be of length `GrB_NAMESIZE`  
980 + 1.

### 981 3.6.2 Hints

982 Several fields are *hints* (marked H in 3.11). A GraphBLAS implementation is free to ignore a  
983 hint and return `GrB_SUCCESS`. For instance `GrB_NTHREADS` might be ignored by a sequential  
984 GraphBLAS implementation.

### 985 3.6.3 get after set

986 If a field allows writes, or `set`, `get` must return the most recent value written. If a field is a *hint*,  
987 the hint is returned, not the value used by the library internally.

988 An exception to this is fields which return the `GrB_ALREADY_SET` return code if they are set  
989 more than once. In Tables 3.11, `GrB_NAME` for Algebraic types and `GrB_Type` may only be set  
990 once, and will return `GrB_ALREADY_SET` on subsequent calls to `GrB_set`.

Table 3.11: Field values of type GrB\_Field enumeration, corresponding types, and the objects which must implement that GrB\_Field. Collection refers to GrB\_Matrix, GrB\_Vector, and GrB\_Scalar, Algebraic refers to Operators, Monoids, and Semirings, while All refers to all GraphBLAS objects. Global fields are denoted by Global. All fields may be read, some may be written (denoted by W), and some are hints (denoted by H) which may be ignored by the implementation. For \* see 3.6.3

(a) Types used with GraphBLAS descriptors.

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

## 3.7 Descriptors

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

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

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

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

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

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

## 3.8 GrB\_Info return values

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



---

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

(a) Types used with GraphBLAS descriptors.

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

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

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

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

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

---

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

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

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

(a) Informational return values

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

(b) API errors

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

## 1046 **Return Values**

1047 `GrB_SUCCESS` operation completed successfully.

1048 `GrB_PANIC` unknown internal error.

1049 `GrB_INVALID_VALUE` invalid mode specified, or method called multiple times.

## 1050 **Description**

1051 The `init` method creates and initializes a GraphBLAS C API context. The argument to `GrB_init`  
1052 defines the mode for the context. The two available modes are:

- 1053 • `GrB_BLOCKING`: In this mode, each method in a sequence returns after its computations have  
1054 completed and output arguments are available to subsequent statements in an application.  
1055 When executing in `GrB_BLOCKING` mode, the methods execute in program order.
- 1056 • `GrB_NONBLOCKING`: In this mode, methods in a sequence may return after arguments in  
1057 the method have been tested for dimension and domain compatibility within the method  
1058 but potentially before their computations complete. Output arguments are available to sub-  
1059 sequent GraphBLAS methods in an application. When executing in `GrB_NONBLOCKING`  
1060 mode, the methods in a sequence may execute in any order that preserves the mathematical  
1061 result defined by the sequence.

1062 An application can only create one context per execution instance. An application may only call  
1063 `GrB_Init` once. Calling `GrB_Init` more than once results in undefined behavior.

### 1064 **4.1.2 finalize: Finalize a GraphBLAS context**

1065 Terminates and frees any internal resources created to support the GraphBLAS C API context.

## 1066 **C Syntax**

1067 `GrB_Info GrB_finalize();`

## 1068 **Return Values**

1069 `GrB_SUCCESS` operation completed successfully.

1070 `GrB_PANIC` unknown internal error.

## 1071 **Description**

1072 The `finalize` method terminates and frees any internal resources created to support the GraphBLAS  
1073 C API context. `GrB_finalize` may only be called after a context has been initialized by calling  
1074 `GrB_init`, or else undefined behavior occurs. After `GrB_finalize` has been called to finalize a Graph-  
1075 BLAS context, calls to any GraphBLAS methods, including `GrB_finalize`, will result in undefined  
1076 behavior.

### 1077 **4.1.3 getVersion: Get the version number of the standard.**

1078 Query the library for the version number of the standard that this library implements.

## 1079 **C Syntax**

```
1080         GrB_Info GrB_getVersion(unsigned int *version,  
1081                                unsigned int *subversion);
```

## 1082 **Parameters**

1083 version (OUT) On successful return will hold the value of the major version number.

1084 version (OUT) On successful return will hold the value of the subversion number.

## 1085 **Return Values**

1086 GrB\_SUCCESS operation completed successfully.

1087 GrB\_PANIC unknown internal error.

## 1088 **Description**

1089 The `getVersion` method is used to query the major and minor version number of the GraphBLAS  
1090 C API specification that the library implements at runtime. To support compile time queries the  
1091 following two macros shall also be defined by the library.

```
1092         #define GRB_VERSION      2  
1093         #define GRB_SUBVERSION  0
```

## 1094 **4.2 Object methods**

1095 This section describes methods that setup and operate on GraphBLAS opaque objects but are not  
1096 part of the the GraphBLAS math specification.

## 1097 4.2.1 Get and Set methods

1098 The methods in this section query and, optionally, set internal fields of GraphBLAS objects.

### 1099 4.2.1.1 get: Query the value of an object

#### 1100 C Syntax

```
1101     GrB_Info GrB_<OBJ>_get(GrB_<OBJ> o, GrB_Field field, <type> value);
1102
1103     GrB_Info GrB_Scalar_get(GrB_Scalar s, GrB_Field field, <type> value);
1104     GrB_Info GrB_Vector_get(GrB_Vector v, GrB_Field field, <type> value);
1105     GrB_Info GrB_Matrix_get(GrB_Matrix A, GrB_Field field, <type> value);
1106
1107     GrB_Info GrB_UnaryOp_get(GrB_UnaryOp op, GrB_Field field, <type> value);
1108     GrB_Info GrB_IndexUnaryOp_get(GrB_IndexUnaryOp op, GrB_Field field, <type> value);
1109     GrB_Info GrB_BinaryOp_get(GrB_BinaryOp op, GrB_Field field, <type> value);
1110     GrB_Info GrB_Monoid_get(GrB_Monoid op, GrB_Field field, <type> value);
1111     GrB_Info GrB_Semiring_get(GrB_Semiring op, GrB_Field field, <type> value);
1112
1113     GrB_Info GrB_Descriptor_get(GrB_Descriptor desc, GrB_Field field, <type> value);
1114     GrB_Info GrB_Type_get(GrB_Type type, GrB_Field field, <type> value);
1115
1116     GrB_Info GrB_Global_get(GrB_Field field, <type> value);
```

#### 1117 Parameters

1118 OBJ is replaced in each signature by the object type being queried.

1119 OBJ (IN) An existing, valid GraphBLAS object which is being queried.

1120 field (IN) The internal field being queried.

1121 value (OUT) A pointer to or GrB\_Scalar containing a value whose type is dependent on  
1122 field which will be filled with the value of the internal field. type may be a char\*,  
1123 int\*, GrB\_Scalar, GrB\_Type, or void\*.

#### 1124 Return Value

1125 GrB\_SUCCESS The method completed successfully.

1126 GrB\_PANIC unknown internal error.

1127 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.



1128 GrB\_UNINITIALIZED\_OBJECT the desc parameter has not been initialized by a call to new.

1129 GrB\_INVALID\_VALUE invalid value set on the field, or invalid field.

## 1130 Description

1131 Queries a field of an existing GraphBLAS object. The type of the variadic argument is uniquely  
1132 determined by field. Fields marked as hints in TABLEREF will return the hint when queried, not  
1133 the true internal value.

### 1134 4.2.1.2 set: Set field of an object

1135 Sets the content for a field for an existing GraphBLAS object.

## 1136 C Syntax

```
1137 GrB_Info GrB_<OBJ>_set(GrB_<OBJ> o, GrB_Field field, <type> value);
1138
1139 GrB_Info GrB_Scalar_set(GrB_Scalar s, GrB_Field field, <type> value);
1140 GrB_Info GrB_Vector_set(GrB_Vector v, GrB_Field field, <type> value);
1141 GrB_Info GrB_Matrix_set(GrB_Matrix A, GrB_Field field, <type> value);
1142
1143 GrB_Info GrB_UnaryOp_set(GrB_UnaryOp op, GrB_Field field, <type> value);
1144 GrB_Info GrB_IndexUnaryOp_set(GrB_IndexUnaryOp op, GrB_Field field, <type> value);
1145 GrB_Info GrB_BinaryOp_set(GrB_BinaryOp op, GrB_Field field, <type> value);
1146 GrB_Info GrB_Monoid_set(GrB_Monoid op, GrB_Field field, <type> value);
1147 GrB_Info GrB_Semiring_set(GrB_Semiring op, GrB_Field field, <type> value);
1148
1149 GrB_Info GrB_Descriptor_set(GrB_Descriptor op, GrB_Field field, <type> value);
1150 GrB_Info GrB_Type_set(GrB_Type op, GrB_Field field, <type> value);
1151
1152 GrB_Info GrB_Global_set(GrB_Field field, <type> value);
```

## 1153 Parameters

1154 OBJ (IN) The GraphBLAS object which is having field set.

1155 field (IN) The field being set.

1156 value (IN) A value whose type is dependent on field or a GrB\_Scalar. type may be a  
1157 char\*, int, GrB\_Scalar, or void\*.

## 1158 **Return Values**

- 1159                   GrB\_SUCCESS The method completed successfully.
- 1160                   GrB\_PANIC unknown internal error.
- 1161           GrB\_OUT\_OF\_MEMORY not enough memory available for operation.
- 1162 GrB\_UNINITIALIZED\_OBJECT the desc parameter has not been initialized by a call to new.
- 1163           GrB\_INVALID\_VALUE invalid value set on the field, or invalid field.
- 1164           GrB\_ALREADY\_SET this field has already been set, and may only be set once.

## 1165 **Description**

1166 Set a field of OBJ to a new value.

## 1167 **4.2.2 Algebra methods**

### 1168 **4.2.2.1 Type\_new: Construct a new GraphBLAS (user-defined) type**

1169 Creates a new user-defined GraphBLAS type. This type can then be used to create new operators,  
1170 monoids, semirings, vectors and matrices.

## 1171 **C Syntax**

```
1172           GrB_Info GrB_Type_new(GrB_Type *utype,  
1173                                   size_t     sizeof(ctype));
```

## 1174 **Parameters**

- 1175           utype (INOUT) On successful return, contains a handle to the newly created user-defined  
1176                   GraphBLAS type object.
- 1177           ctype (IN) A C type that defines the new GraphBLAS user-defined type.

## 1178 **Return Values**

- 1179                   GrB\_SUCCESS operation completed successfully.
- 1180                   GrB\_PANIC unknown internal error.
- 1181           GrB\_OUT\_OF\_MEMORY not enough memory available for operation.
- 1182           GrB\_NULL\_POINTER utype pointer is NULL.

## 1183 Description

1184 Given a C type `ctype`, the `Type_new` method returns in `utype` a handle to a new GraphBLAS type  
1185 that is equivalent to the C type. Variables of this `ctype` must be a struct, union, or fixed-size array.  
1186 In particular, given two variables, `src` and `dst`, of type `ctype`, the following operation must be a  
1187 valid way to copy the contents of `src` to `dst`:

```
1188             memcpy(&dst, &src, sizeof(ctype))
```

1189 A new, user-defined type `utype` should be destroyed with a call to `GrB_free(utype)` when no longer  
1190 needed.

1191 It is not an error to call this method more than once on the same variable; however, the handle to  
1192 the previously created object will be overwritten.

### 1193 4.2.2.2 UnaryOp\_new: Construct a new GraphBLAS unary operator

1194 Initializes a new GraphBLAS unary operator with a specified user-defined function and its types  
1195 (domains).

## 1196 C Syntax

```
1197     GrB_Info GrB_UnaryOp_new(GrB_UnaryOp *unary_op,  
1198                             void          (*unary_func)(void*, const void*),  
1199                             GrB_Type      d_out,  
1200                             GrB_Type      d_in);
```

## 1201 Parameters

1202 `unary_op` (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1203 unary operator object.

1204 `unary_func` (IN) a pointer to a user-defined function that takes one input parameter of `d_in`'s  
1205 type and returns a value of `d_out`'s type, both passed as `void` pointers. Specifically  
1206 the signature of the function is expected to be of the form:

```
1207             void func(void *out, const void *in);  
1208
```

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

1212 `d_in` (IN) The `GrB_Type` of the input argument of the unary operator being created.  
1213 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1214 GraphBLAS type.

## 1215 **Return Values**

1216 `GrB_SUCCESS` operation completed successfully.

1217 `GrB_PANIC` unknown internal error.

1218 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1219 `GrB_UNINITIALIZED_OBJECT` any `GrB_Type` parameter (for user-defined types) has not been ini-  
1220 tialized by a call to `GrB_Type_new`.

1221 `GrB_NULL_POINTER` `unary_op` or `unary_func` pointers are `NULL`.

## 1222 **Description**

1223 The `UnaryOp_new` method creates a new GraphBLAS unary operator

1224  $f_u = \langle \mathbf{D}(\mathbf{d\_out}), \mathbf{D}(\mathbf{d\_in}), \text{unary\_func} \rangle$

1225 and returns a handle to it in `unary_op`.

1226 The implementation of `unary_func` must be such that it works even if the `d_out` and `d_in` arguments  
1227 are aliased. In other words, for all invocations of the function:

1228 `unary_func(out,in);`

1229 the value of `out` must be the same as if the following code was executed:

```
1230 D(d_in) *tmp = malloc(sizeof(D(d_in)));  
1231 memcpy(tmp,in,sizeof(D(d_in)));  
1232 unary_func(out,tmp);  
1233 free(tmp);
```

1234 It is not an error to call this method more than once on the same variable; however, the handle to  
1235 the previously created object will be overwritten.

### 1236 **4.2.2.3 BinaryOp\_new: Construct a new GraphBLAS binary operator**

1237 Initializes a new GraphBLAS binary operator with a specified user-defined function and its types  
1238 (domains).

## 1239 **C Syntax**

```
1240 GrB_Info GrB_BinaryOp_new(GrB_BinaryOp *binary_op,  
1241 void (*binary_func)(void*,
```

```

1242                                     const void*,
1243                                     const void*),
1244         GrB_Type      d_out,
1245         GrB_Type      d_in1,
1246         GrB_Type      d_in2);

```

## 1247 Parameters

1248     **binary\_op** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1249     binary operator object.

1250     **binary\_func** (IN) A pointer to a user-defined function that takes two input parameters of types  
1251     **d\_in1** and **d\_in2** and returns a value of type **d\_out**, all passed as void pointers.  
1252     Specifically the signature of the function is expected to be of the form:

```

1253         void func(void *out, const void *in1, const void *in2);
1254

```

1255     **d\_out** (IN) The **GrB\_Type** of the return value of the binary operator being created. Should  
1256     be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-  
1257     BLAS type.

1258     **d\_in1** (IN) The **GrB\_Type** of the left hand argument of the binary operator being created.  
1259     Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1260     GraphBLAS type.

1261     **d\_in2** (IN) The **GrB\_Type** of the right hand argument of the binary operator being cre-  
1262     ated. Should be one of the predefined GraphBLAS types in Table 3.2, or a user-  
1263     defined GraphBLAS type.

## 1264 Return Values

1265     **GrB\_SUCCESS** operation completed successfully.

1266     **GrB\_PANIC** unknown internal error.

1267     **GrB\_OUT\_OF\_MEMORY** not enough memory available for operation.

1268     **GrB\_UNINITIALIZED\_OBJECT** the **GrB\_Type** (for user-defined types) has not been initialized by a  
1269     call to **GrB\_Type\_new**.

1270     **GrB\_NULL\_POINTER** **binary\_op** or **binary\_func** pointer is NULL.

## 1271 Description

1272     The **BinaryOp\_new** methods creates a new GraphBLAS binary operator

1273  $f_b = \langle \mathbf{D}(d\_out), \mathbf{D}(d\_in1), \mathbf{D}(d\_in2), \text{binary\_func} \rangle$

1274 and returns a handle to it in `binary_op`.

1275 The implementation of `binary_func` must be such that it works even if any of the `d_out`, `d_in1`, and  
1276 `d_in2` arguments are aliased to each other. In other words, for all invocations of the function:

1277 `binary_func(out, in1, in2);`

1278 the value of `out` must be the same as if the following code was executed:

```
1279 D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));  
1280 D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));  
1281 memcpy(tmp1, in1, sizeof(D(d_in1)));  
1282 memcpy(tmp2, in2, sizeof(D(d_in2)));  
1283 binary_func(out, tmp1, tmp2);  
1284 free(tmp2);  
1285 free(tmp1);
```

1286 It is not an error to call this method more than once on the same variable; however, the handle to  
1287 the previously created object will be overwritten.

#### 1288 **4.2.2.4 Monoid\_new: Construct a new GraphBLAS monoid**

1289 Creates a new monoid with specified binary operator and identity value.

### 1290 **C Syntax**

```
1291 GrB_Info GrB_Monoid_new(GrB_Monoid *monoid,  
1292                         GrB_BinaryOp binary_op,  
1293                         <type> identity);
```

### 1294 **Parameters**

1295 `monoid` (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1296 monoid object.

1297 `binary_op` (IN) An existing GraphBLAS associative binary operator whose input and output  
1298 types are the same.

1299 `identity` (IN) The value of the identity element of the monoid. Must be the same type as  
1300 the type used by the `binary_op` operator.

## 1301 Return Values

1302                   GrB\_SUCCESS operation completed successfully.

1303                   GrB\_PANIC unknown internal error.

1304                   GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1305 GrB\_UNINITIALIZED\_OBJECT the GrB\_BinaryOp (for user-defined operators) has not been initial-  
1306                   ized by a call to GrB\_BinaryOp\_new.

1307                   GrB\_NULL\_POINTER monoid pointer is NULL.

1308                   GrB\_DOMAIN\_MISMATCH all three argument types of the binary operator and the type of the  
1309                   identity value are not the same.

## 1310 Description

1311 The Monoid\_new method creates a new monoid  $M = \langle \mathbf{D}(\text{binary\_op}), \text{binary\_op}, \text{identity} \rangle$  and re-  
1312 turns a handle to it in monoid.

1313 If binary\_op is not associative, the results of GraphBLAS operations that require associativity of  
1314 this monoid will be undefined.

1315 It is not an error to call this method more than once on the same variable; however, the handle to  
1316 the previously created object will be overwritten.

### 1317 4.2.2.5 Semiring\_new: Construct a new GraphBLAS semiring

1318 Creates a new semiring with specified domain, operators, and elements.

## 1319 C Syntax

```
1320                   GrB_Info GrB_Semiring_new(GrB_Semiring *semiring,  
1321                                           GrB_Monoid       add_op,  
1322                                           GrB_BinaryOp   mul_op);
```

## 1323 Parameters

1324                   semiring (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1325                   semiring.

1326                   add\_op (IN) An existing GraphBLAS commutative monoid that specifies the addition op-  
1327                   erator and its identity.

1328                   mul\_op (IN) An existing GraphBLAS binary operator that specifies the semiring's multi-  
1329                   plication operator. In addition, mul\_op's output domain,  $\mathbf{D}_{out}(\text{mul\_op})$ , must be  
1330                   the same as the add\_op's domain  $\mathbf{D}(\text{add\_op})$ .

## 1331 Return Values

1332           GrB\_SUCCESS operation completed successfully.

1333           GrB\_PANIC unknown internal error.

1334           GrB\_OUT\_OF\_MEMORY not enough memory available for this method to complete.

1335 GrB\_UNINITIALIZED\_OBJECT the add\_op (for user-define monoids) object has not been initialized  
1336                           with a call to GrB\_Monoid\_new or the mul\_op (for user-defined  
1337                           operators) object has not been not been initialized by a call to  
1338                           GrB\_BinaryOp\_new.

1339           GrB\_NULL\_POINTER semiring pointer is NULL.

1340           GrB\_DOMAIN\_MISMATCH the output domain of mul\_op does not match the domain of the  
1341                           add\_op monoid.

## 1342 Description

1343 The Semiring\_new method creates a new semiring:

1344            $S = \langle \mathbf{D}_{out}(\text{mul\_op}), \mathbf{D}_{in_1}(\text{mul\_op}), \mathbf{D}_{in_2}(\text{mul\_op}), \text{add\_op}, \text{mul\_op}, \mathbf{0}(\text{add\_op}) \rangle$

1345 and returns a handle to it in semiring. Note that  $\mathbf{D}_{out}(\text{mul\_op})$  must be the same as  $\mathbf{D}(\text{add\_op})$ .

1346 If add\_op is not commutative, then GraphBLAS operations using this semiring will be undefined.

1347 It is not an error to call this method more than once on the same variable; however, the handle to  
1348 the previously created object will be overwritten.

### 1349 4.2.2.6 IndexUnaryOp\_new: Construct a new GraphBLAS index unary operator [Scott: 1350 NEW CONTENT]

1351 Initializes a new GraphBLAS index unary operator with a specified user-defined function and its  
1352 types (domains).

## 1353 C Syntax

```
1354 GrB_Info GrB_IndexUnaryOp_new(GrB_IndexUnaryOp *index_unary_op,  
1355                               void (*index_unary_func)(void*,  
1356                                                         const void*,  
1357                                                         GrB_Index,  
1358                                                         GrB_Index,  
1359                                                         const void*),  
1360                               GrB_Type d_out,  
1361                               GrB_Type d_in1,  
1362                               GrB_Type d_in2);
```



## 1363 Parameters

1364 **index\_unary\_op** (INOUT) On successful return, contains a handle to the newly created Graph-  
1365 BLAS index unary operator object.

1366 **index\_unary\_func** (IN) A pointer to a user-defined function that takes input parameters of types  
1367 **d\_in1**, **GrB\_Index**, **GrB\_Index** and **d\_in2** and returns a value of type **d\_out**. Ex-  
1368 cept for the **GrB\_Index** parameters, all are passed as **void** pointers. Specifically  
1369 the signature of the function is expected to be of the form:

```
1370         void func(void      *out,  
1371                   const void *in1,  
1372                   GrB_Index  row_index,  
1373                   GrB_Index  col_index,  
1374                   const void *in2);
```

1375

1376 **d\_out** (IN) The **GrB\_Type** of the return value of the index unary operator being created.  
1377 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1378 GraphBLAS type.

1379 **d\_in1** (IN) The **GrB\_Type** of the first input argument of the index unary operator being  
1380 created and corresponds to the stored values of the **GrB\_Vector** or **GrB\_Matrix**  
1381 being operated on. Should be one of the predefined GraphBLAS types in Ta-  
1382 ble 3.2, or a user-defined GraphBLAS type.

1383 **d\_in2** (IN) The **GrB\_Type** of the last input argument of the index unary operator be-  
1384 ing created and corresponds to a scalar provided by the GraphBLAS operation  
1385 that uses this operator. Should be one of the predefined GraphBLAS types in  
1386 Table 3.2, or a user-defined GraphBLAS type.

## 1387 Return Values

1388 **GrB\_SUCCESS** operation completed successfully.

1389 **GrB\_PANIC** unknown internal error.

1390 **GrB\_OUT\_OF\_MEMORY** not enough memory available for operation.

1391 **GrB\_UNINITIALIZED\_OBJECT** the **GrB\_Type** (for user-defined types) has not been initialized by a  
1392 call to **GrB\_Type\_new**.

1393 **GrB\_NULL\_POINTER** **index\_unary\_op** or **index\_unary\_func** pointer is **NULL**.

## 1394 Description

1395 The **IndexUnaryOp\_new** methods creates a new GraphBLAS index unary operator

1396  $f_i = \langle \mathbf{D}(\mathbf{d\_out}), \mathbf{D}(\mathbf{d\_in1}), \mathbf{D}(\mathbf{GrB\_Index}), \mathbf{D}(\mathbf{GrB\_Index}), \mathbf{D}(\mathbf{d\_in2}), \text{index\_unary\_func} \rangle$

1397 and returns a handle to it in `index_unary_op`.

1398 The implementation of `index_unary_func` must be such that it works even if any of the `d_out`,  
 1399 `d_in1`, and `d_in2` arguments are aliased to each other. In other words, for all invocations of the  
 1400 function:

1401 `index_unary_func(out, in1, row_index, col_index, n, in2);`

1402 the value of `out` must be the same as if the following code was executed (shown here for matrices):

```
1403 GrB_Index row_index = ...;
1404 GrB_Index col_index = ...;
1405 D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));
1406 D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));
1407 memcpy(tmp1, in1, sizeof(D(d_in1)));
1408 memcpy(tmp2, in2, sizeof(D(d_in2)));
1409 index_unary_func(out, tmp1, row_index, col_index, tmp2);
1410 free(tmp2);
1411 free(tmp1);
```

1412 It is not an error to call this method more than once on the same variable; however, the handle to  
 1413 the previously created object will be overwritten.

## 1414 4.2.3 Scalar methods

### 1415 4.2.3.1 Scalar\_new: Construct a new scalar

1416 Creates a new empty scalar with specified domain.

## 1417 C Syntax

```
1418 GrB_Info GrB_Scalar_new(GrB_Scalar *s,
1419                          GrB_Type d);
```

## 1420 Parameters

1421 **s** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
 1422 scalar.

1423 **d** (IN) The type corresponding to the domain of the scalar being created. Can be  
 1424 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
 1425 GraphBLAS type.

## 1426 Return Values

1427           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1428                       blocking mode, this indicates that the API checks for the input  
1429                       arguments passed successfully. Either way, output scalar `s` is ready  
1430                       to be used in the next method of the sequence.

1431           GrB\_PANIC Unknown internal error.

1432           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1433                       GraphBLAS objects (input or output) is in an invalid state caused  
1434                       by a previous execution error. Call `GrB_error()` to access any error  
1435                       messages generated by the implementation.

1436           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1437 GrB\_UNINITIALIZED\_OBJECT The `GrB_Type` object has not been initialized by a call to `GrB_Type_new`  
1438                       (needed for user-defined types).

1439           GrB\_NULL\_POINTER The `s` pointer is NULL.

## 1440 Description

1441 Creates a new GraphBLAS scalar `s` of domain `D(d)` and empty `L(s)`. The method returns a handle  
1442 to the new scalar in `s`.

1443 It is not an error to call this method more than once on the same variable; however, the handle to  
1444 the previously created object will be overwritten.

### 1445 4.2.3.2 Scalar\_dup: Construct a copy of a GraphBLAS scalar

1446 Creates a new scalar with the same domain and contents as another scalar.

## 1447 C Syntax

```
1448           GrB_Info GrB_Scalar_dup(GrB_Scalar           *t,  
1449                                   const GrB_Scalar   s);
```

## 1450 Parameters

1451           t (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1452           scalar.

1453           s (IN) The GraphBLAS scalar to be duplicated.

## 1454 Return Values

1455           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1456                       blocking mode, this indicates that the API checks for the input  
1457                       arguments passed successfully. Either way, output scalar *t* is ready  
1458                       to be used in the next method of the sequence.

1459           GrB\_PANIC Unknown internal error.

1460           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1461                       GraphBLAS objects (input or output) is in an invalid state caused  
1462                       by a previous execution error. Call `GrB_error()` to access any error  
1463                       messages generated by the implementation.

1464           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1465           GrB\_UNINITIALIZED\_OBJECT The GraphBLAS scalar, *s*, has not been initialized by a call to  
1466                       `Scalar_new` or `Scalar_dup`.

1467           GrB\_NULL\_POINTER The *t* pointer is NULL.

## 1468 Description

1469   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  
1470   scalar in *t*.

1471   It is not an error to call this method more than once with the same output variable; however, the  
1472   handle to the previously created object will be overwritten.

### 1473 4.2.3.3 Scalar\_clear: Clear/remove a stored value from a scalar

1474   Removes the stored value from a scalar.

## 1475 C Syntax

1476           GrB\_Info GrB\_Scalar\_clear(GrB\_Scalar s);

## 1477 Parameters

1478           *s* (INOUT) An existing GraphBLAS scalar to clear.

## 1479 Return Values

1480           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1481                       blocking mode, this indicates that the API checks for the input

1482 arguments passed successfully. Either way, output scalar `s` is ready  
 1483 to be used in the next method of the sequence.

1484 `GrB_PANIC` Unknown internal error.

1485 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque  
 1486 GraphBLAS objects (input or output) is in an invalid state caused  
 1487 by a previous execution error. Call `GrB_error()` to access any error  
 1488 messages generated by the implementation.

1489 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1490 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS scalar, `s`, has not been initialized by a call to  
 1491 `Scalar_new` or `Scalar_dup`.

## 1492 Description

1493 Removes the stored value from an existing scalar. After the call, `L(s)` is empty. The size of the  
 1494 scalar does not change.

### 1495 4.2.3.4 `Scalar_nvals`: Number of stored elements in a scalar

1496 Retrieve the number of stored elements in a scalar (either zero or one).

## 1497 C Syntax

```
1498 GrB_Info GrB_Scalar_nvals(GrB_Index      *nvals,
1499                          const GrB_Scalar s);
```

## 1500 Parameters

1501 `nvals` (OUT) On successful return, this is set to the number of stored elements in the  
 1502 scalar (zero or one).

1503 `s` (IN) An existing GraphBLAS scalar being queried.

## 1504 Return Values

1505 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-  
 1506 cessfully and the value of `nvals` has been set.

1507 `GrB_PANIC` Unknown internal error.

1508           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 1509                                   GraphBLAS objects (input or output) is in an invalid state caused  
 1510                                   by a previous execution error. Call GrB\_error() to access any error  
 1511                                   messages generated by the implementation.

1512           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1513 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS scalar, *s*, has not been initialized by a call to  
 1514                                   Scalar\_new or Scalar\_dup.

1515           GrB\_NULL\_POINTER The *nvals* pointer is NULL.

## 1516 Description

1517 Return *nvals(s)* in *nvals*. This is the number of stored elements in scalar *s*, which is the size of  
 1518 *L(s)*, and can only be either zero or one (see Section 3.5.1).

### 1519 4.2.3.5 Scalar\_setElement: Set the single element in a scalar

1520 Set the single element of a scalar to a given value.

## 1521 C Syntax

```
1522           GrB_Info GrB_Scalar_setElement(GrB_Scalar    s,
1523                                                           <type>    val);
```

## 1524 Parameters

1525           *s* (INOUT) An existing GraphBLAS scalar for which the element is to be assigned.

1526           *val* (IN) Scalar value to assign. The type must be compatible with the domain of *s*.

## 1527 Return Values

1528           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 1529                                   blocking mode, this indicates that the compatibility tests on in-  
 1530                                   dex/dimensions and domains for the input arguments passed suc-  
 1531                                   cessfully. Either way, the output scalar *s* is ready to be used in the  
 1532                                   next method of the sequence.

1533           GrB\_PANIC Unknown internal error.

1534           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 1535                                   GraphBLAS objects (input or output) is in an invalid state caused

1536 by a previous execution error. Call `GrB_error()` to access any error  
1537 messages generated by the implementation.

1538 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1539 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS scalar, `s`, has not been initialized by a call to  
1540 `Scalar_new` or `Scalar_dup`.

1541 **GrB\_DOMAIN\_MISMATCH** The domains of `s` and `val` are incompatible.

## 1542 Description

1543 First, `val` and output GraphBLAS scalar are tested for domain compatibility as follows: **D**(`val`) must  
1544 be compatible with **D**(`s`). Two domains are compatible with each other if values from one domain  
1545 can be cast to values in the other domain as per the rules of the C language. In particular, domains  
1546 from Table 3.2 are all compatible with each other. A domain from a user-defined type is only com-  
1547 patible with itself. If any compatibility rule above is violated, execution of `GrB_Scalar_setElement`  
1548 ends and the domain mismatch error listed above is returned.

1549 We are now ready to carry out the assignment `val`; that is:

$$1550 \quad s(0) = val$$

1551 If `s` already had a stored value, it will be overwritten; otherwise, the new value is stored in `s`.

1552 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents  
1553 of `s` is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with  
1554 return value `GrB_SUCCESS` and the new content of scalar `s` is as defined above but may not be  
1555 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 1556 4.2.3.6 `Scalar_extractElement`: Extract a single element from a scalar.

1557 Assign a non-opaque scalar with the value of the element stored in a GraphBLAS scalar.

## 1558 C Syntax

```
1559      GrB_Info GrB_Scalar_extractElement(<type>          *val,  
1560                                         const GrB_Scalar s);
```

## 1561 Parameters

1562 `val` (INOUT) Pointer to a non-opaque scalar of type that is compatible with the domain  
1563 of scalar `s`. On successful return, `val` holds the result of the operation, and any  
1564 previous value in `val` is overwritten.

1565 `s` (IN) The GraphBLAS scalar from which an element is extracted.

## 1566 Return Values

1567           **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
1568                           cessfully. This indicates that the compatibility tests on dimensions  
1569                           and domains for the input arguments passed successfully, and the  
1570                           output scalar, `val`, has been computed and is ready to be used in  
1571                           the next method of the sequence.

1572           **GrB\_PANIC** Unknown internal error.

1573           **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1574                           GraphBLAS objects (input or output) is in an invalid state caused  
1575                           by a previous execution error. Call `GrB_error()` to access any error  
1576                           messages generated by the implementation.

1577           **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1578 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS scalar, `s`, has not been initialized by a call to  
1579                           `Scalar_new` or `Scalar_dup`.

1580           **GrB\_NULL\_POINTER** `val` pointer is NULL.

1581           **GrB\_DOMAIN\_MISMATCH** The domains of the scalar or scalar are incompatible.

1582           **GrB\_NO\_VALUE** There is no stored value in the scalar.

## 1583 Description

1584 First, `val` and input GraphBLAS scalar are tested for domain compatibility as follows: **D(val)**  
1585 must be compatible with **D(s)**. Two domains are compatible with each other if values from  
1586 one domain can be cast to values in the other domain as per the rules of the C language. In  
1587 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
1588 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
1589 `GrB_Scalar_extractElement` ends and the domain mismatch error listed above is returned.

1590 Then, if no value is currently stored in the GraphBLAS scalar, the method returns **GrB\_NO\_VALUE**  
1591 and `val` remains unchanged.

1592 Finally the extract into the output argument, `val` can be performed; that is:

1593   
$$\text{val} = \text{s}(0)$$

1594 In both **GrB\_BLOCKING** mode **GrB\_NONBLOCKING** mode if the method exits with return value  
1595 **GrB\_SUCCESS**, the new contents of `val` are as defined above.



## 1596 4.2.4 Vector methods

### 1597 4.2.4.1 Vector\_new: Construct new vector

1598 Creates a new vector with specified domain and size.

## 1599 C Syntax

```
1600         GrB_Info GrB_Vector_new(GrB_Vector *v,  
1601                                GrB_Type    d,  
1602                                GrB_Index   nsize);
```

## 1603 Parameters

1604 v (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1605 vector.

1606 d (IN) The type corresponding to the domain of the vector being created. Can be  
1607 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
1608 GraphBLAS type.

1609 nsize (IN) The size of the vector being created.

## 1610 Return Values

1611 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1612 blocking mode, this indicates that the API checks for the input  
1613 arguments passed successfully. Either way, output vector v is ready  
1614 to be used in the next method of the sequence.

1615 GrB\_PANIC Unknown internal error.

1616 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1617 GraphBLAS objects (input or output) is in an invalid state caused  
1618 by a previous execution error. Call GrB\_error() to access any error  
1619 messages generated by the implementation.

1620 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1621 GrB\_UNINITIALIZED\_OBJECT The GrB\_Type object has not been initialized by a call to GrB\_Type\_new  
1622 (needed for user-defined types).

1623 GrB\_NULL\_POINTER The v pointer is NULL.

1624 GrB\_INVALID\_VALUE nsize is zero or outside the range of the type GrB\_Index.

## 1625 Description

1626 Creates a new vector  $\mathbf{v}$  of domain  $\mathbf{D(d)}$ , size  $\mathbf{nsz}$ , and empty  $\mathbf{L(v)}$ . The method returns a handle  
1627 to the new vector in  $\mathbf{v}$ .

1628 It is not an error to call this method more than once on the same variable; however, the handle to  
1629 the previously created object will be overwritten.

### 1630 4.2.4.2 Vector\_dup: Construct a copy of a GraphBLAS vector

1631 Creates a new vector with the same domain, size, and contents as another vector.

## 1632 C Syntax

```
1633         GrB_Info GrB_Vector_dup(GrB_Vector      *w,  
1634                                const GrB_Vector  u);
```

## 1635 Parameters

1636  $\mathbf{w}$  (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1637 vector.

1638  $\mathbf{u}$  (IN) The GraphBLAS vector to be duplicated.

## 1639 Return Values

1640  $\mathbf{GrB\_SUCCESS}$  In blocking mode, the operation completed successfully. In non-  
1641 blocking mode, this indicates that the API checks for the input  
1642 arguments passed successfully. Either way, output vector  $\mathbf{w}$  is ready  
1643 to be used in the next method of the sequence.

1644  $\mathbf{GrB\_PANIC}$  Unknown internal error.

1645  $\mathbf{GrB\_INVALID\_OBJECT}$  This is returned in any execution mode whenever one of the opaque  
1646 GraphBLAS objects (input or output) is in an invalid state caused  
1647 by a previous execution error. Call  $\mathbf{GrB\_error()}$  to access any error  
1648 messages generated by the implementation.

1649  $\mathbf{GrB\_OUT\_OF\_MEMORY}$  Not enough memory available for operation.

1650  $\mathbf{GrB\_UNINITIALIZED\_OBJECT}$  The GraphBLAS vector,  $\mathbf{u}$ , has not been initialized by a call to  
1651  $\mathbf{Vector\_new}$  or  $\mathbf{Vector\_dup}$ .

1652  $\mathbf{GrB\_NULL\_POINTER}$  The  $\mathbf{w}$  pointer is  $\mathbf{NULL}$ .

## 1653 Description

1654 Creates a new vector  $\mathbf{w}$  of domain  $\mathbf{D}(\mathbf{u})$ , size  $\mathbf{size}(\mathbf{u})$ , and contents  $\mathbf{L}(\mathbf{u})$ . The method returns a  
1655 handle to the new vector in  $\mathbf{w}$ .

1656 It is not an error to call this method more than once on the same variable; however, the handle to  
1657 the previously created object will be overwritten.

### 1658 4.2.4.3 Vector\_resize: Resize a vector

1659 Changes the size of an existing vector.

## 1660 C Syntax

```
1661         GrB_Info GrB_Vector_resize(GrB_Vector  w,  
1662                                   GrB_Index   nsize);
```

## 1663 Parameters

1664  $\mathbf{w}$  (INOUT) An existing Vector object that is being resized.

1665  $\mathbf{nsize}$  (IN) The new size of the vector. It can be smaller or larger than the current size.

## 1666 Return Values

1667 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1668 blocking mode, this indicates that the API checks for the input  
1669 arguments passed successfully. Either way, output vector  $\mathbf{w}$  is ready  
1670 to be used in the next method of the sequence.

1671 GrB\_PANIC Unknown internal error.

1672 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1673 GraphBLAS objects (input or output) is in an invalid state caused  
1674 by a previous execution error. Call GrB\_error() to access any error  
1675 messages generated by the implementation.

1676 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1677 GrB\_NULL\_POINTER The  $\mathbf{w}$  pointer is NULL.

1678 GrB\_INVALID\_VALUE  $\mathbf{nsize}$  is zero or outside the range of the type GrB\_Index.

## 1679 Description

1680 Changes the size of  $w$  to  $nsz$ . The domain  $\mathbf{D}(w)$  of vector  $w$  remains the same. The contents  $\mathbf{L}(w)$   
1681 are modified as described below.

1682 Let  $w = \langle \mathbf{D}(w), N, \mathbf{L}(w) \rangle$  when the method is called. When the method returns,  $w = \langle \mathbf{D}(w), nsz, \mathbf{L}'(w) \rangle$   
1683 where  $\mathbf{L}'(w) = \{(i, w_i) : (i, w_i) \in \mathbf{L}(w) \wedge (i < nsz)\}$ . That is, all elements of  $w$  with index greater  
1684 than or equal to the new vector size ( $nsz$ ) are dropped.

### 1685 4.2.4.4 Vector\_clear: Clear a vector

1686 Removes all the elements (tuples) from a vector.

## 1687 C Syntax

```
1688 GrB_Info GrB_Vector_clear(GrB_Vector v);
```

## 1689 Parameters

1690  $v$  (INOUT) An existing GraphBLAS vector to clear.

## 1691 Return Values

1692 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1693 blocking mode, this indicates that the API checks for the input  
1694 arguments passed successfully. Either way, output vector  $v$  is ready  
1695 to be used in the next method of the sequence.

1696 **GrB\_PANIC** Unknown internal error.

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

1701 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1702 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector,  $v$ , has not been initialized by a call to  
1703 `Vector_new` or `Vector_dup`.

## 1704 Description

1705 Removes all elements (tuples) from an existing vector. After the call to `GrB_Vector_clear(v)`,  
1706  $\mathbf{L}(v) = \emptyset$ . The size of the vector does not change.

#### 1707 **4.2.4.5 Vector\_size: Size of a vector**

1708 Retrieve the size of a vector.

#### 1709 **C Syntax**

```
1710         GrB_Info GrB_Vector_size(GrB_Index      *nsize,  
1711                                   const GrB_Vector v);
```

#### 1712 **Parameters**

1713 nsize (OUT) On successful return, is set to the size of the vector.

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

#### 1715 **Return Values**

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

1718 GrB\_PANIC Unknown internal error.

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

1723 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, v, has not been initialized by a call to  
1724 Vector\_new or Vector\_dup.

1725 GrB\_NULL\_POINTER nsize pointer is NULL.

#### 1726 **Description**

1727 Return **size**(v) in nsize.

#### 1728 **4.2.4.6 Vector\_nvals: Number of stored elements in a vector**

1729 Retrieve the number of stored elements (tuples) in a vector.

#### 1730 **C Syntax**

```
1731         GrB_Info GrB_Vector_nvals(GrB_Index      *nvals,  
1732                                   const GrB_Vector v);
```

## 1733 Parameters

1734            **nvals** (OUT) On successful return, this is set to the number of stored elements (tuples)  
1735            in the vector.

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

## 1737 Return Values

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

1740            **GrB\_PANIC** Unknown internal error.

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

1745            **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1746            **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **v**, has not been initialized by a call to  
1747            **Vector\_new** or **Vector\_dup**.

1748            **GrB\_NULL\_POINTER** The **nvals** pointer is **NULL**.

## 1749 Description

1750            Return **nvals(v)** in **nvals**. This is the number of stored elements in vector **v**, which is the size of  
1751            **L(v)** (see Section 3.5.2).

### 1752 4.2.4.7 Vector\_build: Store elements from tuples into a vector

## 1753 C Syntax

```
1754            GrB_Info GrB_Vector_build(GrB_Vector            w,  
1755                                        const GrB_Index        *indices,  
1756                                        const <type>           *values,  
1757                                        GrB_Index                n,  
1758                                        const GrB_BinaryOp       dup);
```

## 1759 Parameters

1760            **w** (INOUT) An existing Vector object to store the result.

1761            **indices** (IN) Pointer to an array of indices.

1762        **values** (IN) Pointer to an array of scalars of a type that is compatible with the domain of  
 1763        vector **w**.

1764        **n** (IN) The number of entries contained in each array (the same for indices and values).

1765        **dup** (IN) An associative and commutative binary operator to apply when duplicate  
 1766        values for the same location are present in the input arrays. All three domains of  
 1767        **dup** must be the same; hence  $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$ . If **dup** is **GrB\_NULL**,  
 1768        then duplicate locations will result in an error.

## 1769 Return Values

1770        **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 1771        blocking mode, this indicates that the API checks for the input  
 1772        arguments passed successfully. Either way, output vector **w** is  
 1773        ready to be used in the next method of the sequence.

1774        **GrB\_PANIC** Unknown internal error.

1775        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
 1776        opaque GraphBLAS objects (input or output) is in an invalid  
 1777        state caused by a previous execution error. Call **GrB\_error()** to  
 1778        access any error messages generated by the implementation.

1779        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1780        **GrB\_UNINITIALIZED\_OBJECT** Either **w** has not been initialized by a call to **GrB\_Vector\_new**  
 1781        or by **GrB\_Vector\_dup**, or **dup** has not been initialized by a call  
 1782        to **GrB\_BinaryOp\_new**.

1783        **GrB\_NULL\_POINTER** indices or values pointer is **NULL**.

1784        **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in indices is outside the allowed range for **w**.

1785        **GrB\_DOMAIN\_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are  
 1786        not all the same, or the domains of **values** and **w** are incompatible  
 1787        with each other or  $D_{dup}$ .

1788        **GrB\_OUTPUT\_NOT\_EMPTY** Output vector **w** already contains valid tuples (elements). In  
 1789        other words, **GrB\_Vector\_nvals(C)** returns a positive value.

1790        **GrB\_INVALID\_VALUE** indices contains a duplicate location and **dup** is **GrB\_NULL**.

## 1791 Description

1792        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  
 1793        from **w** in its domain; otherwise,  $\tilde{\mathbf{w}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \emptyset \rangle$ .

1794 Each tuple  $\{\text{indices}[k], \text{values}[k]\}$ , where  $0 \leq k < n$ , is a contribution to the output in the form of

$$1795 \quad \tilde{\mathbf{w}}(\text{indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } dup \neq \text{GrB\_NULL} \\ (\mathbf{D}(\mathbf{w})) \text{values}[k] & \text{otherwise.} \end{cases}$$

1796 If multiple values for the same location are present in the input arrays and `dup` is not `GrB_NULL`,  
1797 `dup` is used to reduce the values before assignment into  $\tilde{\mathbf{w}}$  as follows:

$$1798 \quad \tilde{\mathbf{w}}_i = \bigoplus_{k: \text{indices}[k]=i} (D_{dup}) \text{values}[k],$$

1799 where  $\oplus$  is the `dup` binary operator. Finally, the resulting  $\tilde{\mathbf{w}}$  is copied into  $\mathbf{w}$  via typecasting its  
1800 values to  $\mathbf{D}(\mathbf{w})$  if necessary. If  $\oplus$  is not associative or not commutative, the result is undefined.

1801 The nonopaque input arrays, `indices` and `values`, must be at least as large as `n`.

1802 It is an error to call this function on an output object with existing elements. In other words,  
1803 `GrB_Vector_nvals(w)` should evaluate to zero prior to calling this function.

1804 After `GrB_Vector_build` returns, it is safe for a programmer to modify or delete the arrays `indices`  
1805 or `values`.

#### 1806 4.2.4.8 Vector\_setElement: Set a single element in a vector

1807 Set one element of a vector to a given value.

### 1808 C Syntax

```
1809 // scalar value
1810 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1811                               <type>         val,
1812                               GrB_Index       index);
1813
1814 // GraphBLAS scalar
1815 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1816                               const GrB_Scalar s,
1817                               GrB_Index       index);
```

### 1818 Parameters

1819 `w` (INOUT) An existing GraphBLAS vector for which an element is to be assigned.

1820 `val` or `s` (IN) Scalar assign. Its domain (type) must be compatible with the domain of `w`.

1821 `index` (IN) The location of the element to be assigned.



## 1822 Return Values

1823           **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1824                           blocking mode, this indicates that the compatibility tests on in-  
1825                           dex/dimensions and domains for the input arguments passed suc-  
1826                           cessfully. Either way, the output vector **w** is ready to be used in  
1827                           the next method of the sequence.

1828           **GrB\_PANIC** Unknown internal error.

1829           **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1830                           GraphBLAS objects (input or output) is in an invalid state caused  
1831                           by a previous execution error. Call **GrB\_error()** to access any error  
1832                           messages generated by the implementation.

1833           **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1834 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **w**, or GraphBLAS scalar, **s**, has not been  
1835                           initialized by a call to a respective constructor.

1836           **GrB\_INVALID\_INDEX** **index** specifies a location that is outside the dimensions of **w**.

1837           **GrB\_DOMAIN\_MISMATCH** The domains of the vector and the scalar are incompatible.

## 1838 Description

1839 First, the scalar and output vector are tested for domain compatibility as follows: **D(val)** or **D(s)**  
1840 must be compatible with **D(w)**. Two domains are compatible with each other if values from  
1841 one domain can be cast to values in the other domain as per the rules of the C language. In  
1842 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
1843 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
1844 **GrB\_Vector\_setElement** ends and the domain mismatch error listed above is returned.

1845 Then, the **index** parameter is checked for a valid value where the following condition must hold:

$$1846 \qquad 0 \leq \text{index} < \text{size}(\mathbf{w})$$

1847 If this condition is violated, execution of **GrB\_Vector\_setElement** ends and the invalid index error  
1848 listed above is returned.

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

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

1849 In the case of a transparent scalar or if **L(s)** is not empty, then a value will be stored at the  
1850 specified location in **w**, overwriting any value that may have been stored there before. In the case  
1851 of a GraphBLAS scalar, if **L(s)** is empty, then any value stored at the specified location in **w** will  
1852 be removed.

1853 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new contents  
 1854 of **w** is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method exits with  
 1855 return value GrB\_SUCCESS and the new contents of vector **w** is as defined above but may not be  
 1856 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 1857 4.2.4.9 Vector\_removeElement: Remove an element from a vector

1858 Remove (annihilate) one stored element from a vector.

#### 1859 C Syntax

```
1860      GrB_Info GrB_Vector_removeElement(GrB_Vector  w,
1861                                     GrB_Index    index);
```

#### 1862 Parameters

1863 **w** (INOUT) An existing GraphBLAS vector from which an element is to be removed.

1864 **index** (IN) The location of the element to be removed.

#### 1865 Return Values

1866 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 1867 blocking mode, this indicates that the compatibility tests on in-  
 1868 dex/dimensions and domains for the input arguments passed suc-  
 1869 cessfully. Either way, the output vector **w** is ready to be used in  
 1870 the next method of the sequence.

1871 **GrB\_PANIC** Unknown internal error.

1872 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 1873 GraphBLAS objects (input or output) is in an invalid state caused  
 1874 by a previous execution error. Call **GrB\_error()** to access any error  
 1875 messages generated by the implementation.

1876 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1877 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **w**, has not been initialized by a call to  
 1878 **Vector\_new** or **Vector\_dup**.

1879 **GrB\_INVALID\_INDEX** **index** specifies a location that is outside the dimensions of **w**.

## 1880 Description

1881 First, the `index` parameter is checked for a valid value where the following condition must hold:

$$1882 \quad 0 \leq \text{index} < \text{size}(\mathbf{w})$$

1883 If this condition is violated, execution of `GrB_Vector_removeElement` ends and the invalid index  
1884 error listed above is returned.

1885 We are now ready to carry out the removal of a value that may be stored at the location specified  
1886 by `index`. If a value does not exist at the specified location in  $\mathbf{w}$ , no error is reported and the  
1887 operation has no effect on the state of  $\mathbf{w}$ . In either case, the following will be true on return from  
1888 the method: `index`  $\notin$  `ind(w)`.

1889 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents  
1890 of  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with  
1891 return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but may not be  
1892 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 1893 4.2.4.10 Vector\_extractElement: Extract a single element from a vector.

1894 Extract one element of a vector into a scalar.

## 1895 C Syntax

```
1896 // scalar value
1897 GrB_Info GrB_Vector_extractElement(<type>          *val,
1898                                   const GrB_Vector u,
1899                                   GrB_Index         index);
1900
1901 // GraphBLAS scalar
1902 GrB_Info GrB_Vector_extractElement(GrB_Scalar      s,
1903                                   const GrB_Vector u,
1904                                   GrB_Index         index);
```

## 1905 Parameters

1906 `val` or `s` (INOUT) An existing scalar of whose domain is compatible with the domain of vector  
1907 `u`. On successful return, this scalar holds the result of the extract. Any previous  
1908 value stored in `val` or `s` is overwritten.

1909 `u` (IN) The GraphBLAS vector from which an element is extracted.

1910 `index` (IN) The location in `u` to extract.

## 1911 Return Values

1912           GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
 1913                           cessfully. This indicates that the compatibility tests on dimensions  
 1914                           and domains for the input arguments passed successfully, and the  
 1915                           output scalar, **val** or **s**, has been computed and is ready to be used  
 1916                           in the next method of the sequence.

1917           GrB\_NO\_VALUE When using the transparent scalar, **val**, this is returned when there  
 1918                           is no stored value at specified location.

1919           GrB\_PANIC Unknown internal error.

1920           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 1921                           GraphBLAS objects (input or output) is in an invalid state caused  
 1922                           by a previous execution error. Call **GrB\_error()** to access any error  
 1923                           messages generated by the implementation.

1924           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1925 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, **u**, or scalar, **s**, has not been initialized by  
 1926                           a call to a corresponding constructor.

1927           GrB\_NULL\_POINTER **val** pointer is NULL.

1928           GrB\_INVALID\_INDEX **index** specifies a location that is outside the dimensions of **w**.

1929           GrB\_DOMAIN\_MISMATCH The domains of the vector and scalar are incompatible.

## 1930 Description

1931 First, the scalar and input vector are tested for domain compatibility as follows: **D(val)** or **D(s)**  
 1932 must be compatible with **D(u)**. Two domains are compatible with each other if values from  
 1933 one domain can be cast to values in the other domain as per the rules of the C language. In  
 1934 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
 1935 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
 1936 **GrB\_Vector\_extractElement** ends and the domain mismatch error listed above is returned.

1937 Then, the **index** parameter is checked for a valid value where the following condition must hold:

$$1938 \qquad 0 \leq \text{index} < \text{size}(\mathbf{u})$$

1939 If this condition is violated, execution of **GrB\_Vector\_extractElement** ends and the invalid index  
 1940 error listed above is returned.

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

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

1941 If  $\text{index} \in \mathbf{ind}(u)$ , then the corresponding value from  $u$  is copied into  $s$  or  $val$  with casting as  
 1942 necessary. If  $\text{index} \notin \mathbf{ind}(u)$ , then one of the follow occurs depending on output scalar type:

- 1943 • The GraphBLAS scalar,  $s$ , is cleared and `GrB_SUCCESS` is returned.
- 1944 • The non-opaque scalar,  $val$ , is unchanged, and `GrB_NO_VALUE` is returned.

1945 When using the non-opaque scalar variant ( $val$ ) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`  
 1946 mode, the new contents of  $val$  are as defined above if the method exits with return value `GrB_SUCCESS`  
 1947 or `GrB_NO_VALUE`.

1948 When using the GraphBLAS scalar variant ( $s$ ) with a `GrB_SUCCESS` return value, the method  
 1949 exits and the new contents of  $s$  is as defined above and fully computed in `GrB_BLOCKING` mode.  
 1950 In `GrB_NONBLOCKING` mode, the new contents of  $s$  is as defined above but may not be fully  
 1951 computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 1952 4.2.4.11 Vector\_extractTuples: Extract tuples from a vector

1953 Extract the contents of a GraphBLAS vector into non-opaque data structures.

#### 1954 C Syntax

```

1955      GrB_Info GrB_Vector_extractTuples(GrB_Index      *indices,
1956                                     <type>          *values,
1957                                     GrB_Index        *n,
1958                                     const GrB_Vector  v);
1959
```

1960 **indices** (OUT) Pointer to an array of indices that is large enough to hold all of the stored  
 1961 values' indices.

1962 **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of  
 1963 the stored values whose type is compatible with  $\mathbf{D}(v)$ .

1964 **n** (INOUT) Pointer to a value indicating (on input) the number of elements the  
 1965 values and indices arrays can hold. Upon return, it will contain the number of  
 1966 values written to the arrays.

1967 **v** (IN) An existing GraphBLAS vector.

#### 1968 Return Values

1969 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
 1970 cessfully. This indicates that the compatibility tests on the input  
 1971 argument passed successfully, and the output arrays, **indices** and  
 1972 **values**, have been computed.

1973                   GrB\_PANIC Unknown internal error.

1974           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1975                                   GraphBLAS objects (input or output) is in an invalid state caused  
1976                                   by a previous execution error. Call `GrB_error()` to access any error  
1977                                   messages generated by the implementation.

1978           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1979           GrB\_INSUFFICIENT\_SPACE Not enough space in `indices` and `values` (as indicated by the `n` pa-  
1980                                   rameter) to hold all of the tuples that will be extracted.

1981 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, `v`, has not been initialized by a call to  
1982                                   `Vector_new` or `Vector_dup`.

1983           GrB\_NULL\_POINTER `indices`, `values`, or `n` pointer is NULL.

1984           GrB\_DOMAIN\_MISMATCH The domains of the `v` vector or `values` array are incompatible with  
1985                                   one another.

## 1986 Description

1987 This method will extract all the tuples from the GraphBLAS vector `v`. The values associated  
1988 with those tuples are placed in the `values` array and the indices are placed in the `indices` array.  
1989 Both `indices` and `values` must be pre-allocated by the user to have enough space to hold at least  
1990 `GrB_Vector_nvals(v)` elements before calling this function.

1991 Upon return of this function, `n` will be set to the number of values (and indices) copied. Also, the  
1992 entries of `indices` are unique, but not necessarily sorted. Each tuple  $(i, v_i)$  in `v` is unzipped and  
1993 copied into a distinct  $k$ th location in output vectors:

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

1994 where  $0 \leq k < \text{GrB\_Vector\_nvals}(v)$ . No gaps in output vectors are allowed; that is, if `indices[k]`  
1995 and `values[k]` exist upon return, so does `indices[j]` and `values[j]` for all  $j$  such that  $0 \leq j < k$ .

1996 Note that if the value in `n` on input is less than the number of values contained in the vector `v`,  
1997 then a `GrB_INSUFFICIENT_SPACE` error is returned because it is undefined which subset of values  
1998 would be extracted otherwise.

1999 In both `GrB_BLOCKING` mode `GrB_NONBLOCKING` mode if the method exits with return value  
2000 `GrB_SUCCESS`, the new contents of the arrays `indices` and `values` are as defined above.

## 2001 4.2.5 Matrix methods

### 2002 4.2.5.1 `Matrix_new`: Construct new matrix

2003 Creates a new matrix with specified domain and dimensions.

## 2004 C Syntax

```
2005         GrB_Info GrB_Matrix_new(GrB_Matrix *A,  
2006                                 GrB_Type      d,  
2007                                 GrB_Index     nrows,  
2008                                 GrB_Index     ncols);
```

## 2009 Parameters

2010 **A** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2011 matrix.

2012 **d** (IN) The type corresponding to the domain of the matrix being created. Can be  
2013 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
2014 GraphBLAS type.

2015 **nrows** (IN) The number of rows of the matrix being created.

2016 **ncols** (IN) The number of columns of the matrix being created.

## 2017 Return Values

2018 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2019 blocking mode, this indicates that the API checks for the input ar-  
2020 guments passed successfully. Either way, output matrix **A** is ready  
2021 to be used in the next method of the sequence.

2022 **GrB\_PANIC** Unknown internal error.

2023 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2024 GraphBLAS objects (input or output) is in an invalid state caused  
2025 by a previous execution error. Call **GrB\_error()** to access any error  
2026 messages generated by the implementation.

2027 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2028 **GrB\_UNINITIALIZED\_OBJECT** The **GrB\_Type** object has not been initialized by a call to **GrB\_Type\_new**  
2029 (needed for user-defined types).

2030 **GrB\_NULL\_POINTER** The **A** pointer is **NULL**.

2031 **GrB\_INVALID\_VALUE** **nrows** or **ncols** is zero or outside the range of the type **GrB\_Index**.

## 2032 Description

2033 Creates a new matrix **A** of domain **D**(**d**), size **nrows**  $\times$  **ncols**, and empty **L**(**A**). The method returns  
2034 a handle to the new matrix in **A**.

2035 It is not an error to call this method more than once on the same variable; however, the handle to  
2036 the previously created object will be overwritten.

#### 2037 4.2.5.2 Matrix\_dup: Construct a copy of a GraphBLAS matrix

2038 Creates a new matrix with the same domain, dimensions, and contents as another matrix.

#### 2039 C Syntax

```
2040         GrB_Info GrB_Matrix_dup(GrB_Matrix      *C,  
2041                                const GrB_Matrix A);
```

#### 2042 Parameters

2043 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2044 matrix.

2045 A (IN) The GraphBLAS matrix to be duplicated.

#### 2046 Return Values

2047 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2048 blocking mode, this indicates that the API checks for the input  
2049 arguments passed successfully. Either way, output matrix C is ready  
2050 to be used in the next method of the sequence.

2051 GrB\_PANIC Unknown internal error.

2052 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2053 GraphBLAS objects (input or output) is in an invalid state caused  
2054 by a previous execution error. Call GrB\_error() to access any error  
2055 messages generated by the implementation.

2056 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2057 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2058 any matrix constructor.

2059 GrB\_NULL\_POINTER The C pointer is NULL.

#### 2060 Description

2061 Creates a new matrix C of domain D(A), size nrows(A) × ncols(A), and contents L(A). It returns  
2062 a handle to it in C.



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 4.2.5.3 Matrix\_diag: Construct a diagonal GraphBLAS matrix

2066 Creates a new matrix with the same domain and contents as a GrB\_Vector, and square dimensions  
2067 appropriate for placing the contents of the vector along the specified diagonal of the matrix.

#### 2068 C Syntax

```
2069         GrB_Info GrB_Matrix_diag(GrB_Matrix      *C,  
2070                                 const GrB_Vector  v,  
2071                                 int64_t           k);
```

#### 2072 Parameters

2073 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2074 matrix. The matrix is square with each dimension equal to **size(v) + |k|**.

2075 v (IN) The GraphBLAS vector whose contents will be copied to the diagonal of the  
2076 matrix.

2077 k (IN) The diagonal to which the vector is assigned. k = 0 represents the main  
2078 diagonal, k > 0 is above the main diagonal, and k < 0 is below.

#### 2079 Return Values

2080 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2081 blocking mode, this indicates that the API checks for the input  
2082 arguments passed successfully. Either way, output matrix C is ready  
2083 to be used in the next method of the sequence.

2084 GrB\_PANIC Unknown internal error.

2085 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2086 GraphBLAS objects (input or output) is in an invalid state caused  
2087 by a previous execution error. Call GrB\_error() to access any error  
2088 messages generated by the implementation.

2089 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

2090 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, v, has not been initialized by a call to  
2091 Vector\_new or Vector\_dup.

2092 GrB\_NULL\_POINTER The C pointer is NULL.

## 2093 Description

2094 Creates a new matrix **C** of domain **D(v)**, size  $(\mathbf{size}(\mathbf{v}) + |k|) \times (\mathbf{size}(\mathbf{v}) + |k|)$ , and contents

$$2095 \quad \mathbf{L}(\mathbf{C}) = \{(i, i + k, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k \geq 0 \text{ or}$$

$$2096 \quad \mathbf{L}(\mathbf{C}) = \{(i - k, i, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k < 0.$$

2097 It returns a handle to it in **C**. It is not an error to call this method more than once on the same  
2098 variable; however, the handle to the previously created object will be overwritten.

## 2099 4.2.5.4 Matrix\_resize: Resize a matrix

2100 Changes the dimensions of an existing matrix.

## 2101 C Syntax

```
2102      GrB_Info GrB_Matrix_resize(GrB_Matrix C,  
2103                                GrB_Index  nrows,  
2104                                GrB_Index  ncols);
```

## 2105 Parameters

2106 **C** (INOUT) An existing Matrix object that is being resized.

2107 **nrows** (IN) The new number of rows of the matrix. It can be smaller or larger than the  
2108 current number of rows.

2109 **ncols** (IN) The new number of columns of the matrix. It can be smaller or larger than  
2110 the current number of columns.

## 2111 Return Values

2112 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2113 blocking mode, this indicates that the API checks for the input  
2114 arguments passed successfully. Either way, output matrix **C** is ready  
2115 to be used in the next method of the sequence.

2116 **GrB\_PANIC** Unknown internal error.

2117 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2118 GraphBLAS objects (input or output) is in an invalid state caused  
2119 by a previous execution error. Call **GrB\_error()** to access any error  
2120 messages generated by the implementation.

2121 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2122           GrB\_NULL\_POINTER The C pointer is NULL.

2123           GrB\_INVALID\_VALUE nrows or ncols is zero or outside the range of the type GrB\_Index.

## 2124   Description

2125   Changes the number of rows and columns of C to nrows and ncols, respectively. The domain  $\mathbf{D}(\mathbf{C})$   
2126   of matrix C remains the same. The contents  $\mathbf{L}(\mathbf{C})$  are modified as described below.

2127   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  
2128   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  
2129   greater than or equal to ncols are dropped.  
2130

### 2131   4.2.5.5   Matrix\_clear: Clear a matrix

2132   Removes all elements (tuples) from a matrix.

## 2133   C Syntax

2134           GrB\_Info GrB\_Matrix\_clear(GrB\_Matrix A);

## 2135   Parameters

2136           A (IN) An existing GraphBLAS matrix to clear.

## 2137   Return Values

2138           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2139                       blocking mode, this indicates that the API checks for the input ar-  
2140                       guments passed successfully. Either way, output matrix A is ready  
2141                       to be used in the next method of the sequence.

2142           GrB\_PANIC Unknown internal error.

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

2147           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2148           GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2149                               any matrix constructor.

2150 **Description**

2151 Removes all elements (tuples) from an existing matrix. After the call to `GrB_Matrix_clear(A)`,  
2152  $\mathbf{L}(\mathbf{A}) = \emptyset$ . The dimensions of the matrix do not change.

2153 **4.2.5.6 Matrix\_nrows: Number of rows in a matrix**

2154 Retrieve the number of rows in a matrix.

2155 **C Syntax**

```
2156         GrB_Info GrB_Matrix_nrows(GrB_Index      *nrows,  
2157                                   const GrB_Matrix A);
```

2158 **Parameters**

2159 nrows (OUT) On successful return, contains the number of rows in the matrix.

2160 A (IN) An existing GraphBLAS matrix being queried.

2161 **Return Values**

2162 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
2163 cessfully and the value of `nrows` has been set.

2164 GrB\_PANIC Unknown internal error.

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

2169 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to  
2170 any matrix constructor.

2171 GrB\_NULL\_POINTER `nrows` pointer is NULL.

2172 **Description**

2173 Return `nrows(A)` in `nrows` (the number of rows).

2174 **4.2.5.7 Matrix\_ncols: Number of columns in a matrix**

2175 Retrieve the number of columns in a matrix.

## 2176 C Syntax

```
2177         GrB_Info GrB_Matrix_ncols(GrB_Index      *ncols,  
2178                                   const GrB_Matrix A);
```

## 2179 Parameters

2180 ncols (OUT) On successful return, contains the number of columns in the matrix.

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

## 2182 Return Values

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

2185 GrB\_PANIC Unknown internal error.

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

2190 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2191 any matrix constructor.

2192 GrB\_NULL\_POINTER ncols pointer is NULL.

## 2193 Description

2194 Return **ncols(A)** in ncols (the number of columns).

## 2195 4.2.5.8 Matrix\_nvals: Number of stored elements in a matrix

2196 Retrieve the number of stored elements (tuples) in a matrix.

## 2197 C Syntax

```
2198         GrB_Info GrB_Matrix_nvals(GrB_Index      *nvals,  
2199                                   const GrB_Matrix A);
```

2200 **Parameters**

2201            **nvals** (OUT) On successful return, contains the number of stored elements (tuples) in  
2202            the matrix.

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

2204 **Return Values**

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

2207            **GrB\_PANIC** Unknown internal error.

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

2212            **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2213            **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to  
2214            any matrix constructor.

2215            **GrB\_NULL\_POINTER** The **nvals** pointer is **NULL**.

2216 **Description**

2217 Return **nvals(A)** in **nvals**. This is the number of tuples stored in matrix **A**, which is the size of  
2218 **L(A)** (see Section 3.5.3).

2219 **4.2.5.9 Matrix\_build: Store elements from tuples into a matrix**

2220 **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);
```

2221 **Parameters**

2222            **C** (INOUT) An existing Matrix object to store the result.

2223 **row\_indices** (IN) Pointer to an array of row indices.

2224 **col\_indices** (IN) Pointer to an array of column indices.

2225 **values** (IN) Pointer to an array of scalars of a type that is compatible with the domain of  
2226 matrix, **C**.

2227 **n** (IN) The number of entries contained in each array (the same for **row\_indices**,  
2228 **col\_indices**, and **values**).

2229 **dup** (IN) An associative and commutative binary operator to apply when duplicate  
2230 values for the same location are present in the input arrays. All three domains of  
2231 **dup** must be the same; hence  $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$ . If **dup** is **GrB\_NULL**,  
2232 then duplicate locations will result in an error.

## 2233 Return Values

2234 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2235 blocking mode, this indicates that the API checks for the input  
2236 arguments passed successfully. Either way, output matrix **C** is  
2237 ready to be used in the next method of the sequence.

2238 **GrB\_PANIC** Unknown internal error.

2239 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
2240 opaque GraphBLAS objects (input or output) is in an invalid  
2241 state caused by a previous execution error. Call **GrB\_error()** to  
2242 access any error messages generated by the implementation.

2243 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2244 **GrB\_UNINITIALIZED\_OBJECT** Either **C** has not been initialized by a call to any matrix construc-  
2245 tor, or **dup** has not been initialized by a call to **GrB\_BinaryOp\_new**.

2246 **GrB\_NULL\_POINTER** **row\_indices**, **col\_indices** or **values** pointer is **NULL**.

2247 **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **row\_indices** or **col\_indices** is outside the allowed range  
2248 for **C**.

2249 **GrB\_DOMAIN\_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are  
2250 not all the same, or the domains of **values** and **C** are incompatible  
2251 with each other or  $D_{dup}$ .

2252 **GrB\_OUTPUT\_NOT\_EMPTY** Output matrix **C** already contains valid tuples (elements). In  
2253 other words, **GrB\_Matrix\_nvals(C)** returns a positive value.

2254 **GrB\_INVALID\_VALUE** indices contains a duplicate location and **dup** is **GrB\_NULL**.

## 2255 Description

2256 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  
 2257 only differs from  $\mathbf{C}$  in its domain; otherwise,  $\tilde{\mathbf{C}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \emptyset \rangle$ .

2258 Each tuple  $\{\text{row\_indices}[k], \text{col\_indices}[k], \text{values}[k]\}$ , where  $0 \leq k < n$ , is a contribution to the  
 2259 output in the form of

$$2260 \quad \tilde{\mathbf{C}}(\text{row\_indices}[k], \text{col\_indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \text{dup} \neq \text{GrB\_NULL} \\ (\mathbf{D}(\mathbf{C})) \text{values}[k] & \text{otherwise.} \end{cases}$$

2261 If multiple values for the same location are present in the input arrays and `dup` is not `GrB_NULL`,  
 2262 `dup` is used to reduce the values before assignment into  $\tilde{\mathbf{C}}$  as follows:

$$2263 \quad \tilde{\mathbf{C}}_{ij} = \bigoplus_{k: \text{row\_indices}[k]=i \wedge \text{col\_indices}[k]=j} (D_{dup}) \text{values}[k],$$

2264 where  $\oplus$  is the `dup` binary operator. Finally, the resulting  $\tilde{\mathbf{C}}$  is copied into  $\mathbf{C}$  via typecasting its  
 2265 values to  $\mathbf{D}(\mathbf{C})$  if necessary. If  $\oplus$  is not associative or not commutative, the result is undefined.

2266 The nonopaque input arrays `row_indices`, `col_indices`, and `values` must be at least as large as `n`.

2267 It is an error to call this function on an output object with existing elements. In other words,  
 2268 `GrB_Matrix_nvals(C)` should evaluate to zero prior to calling this function.

2269 After `GrB_Matrix_build` returns, it is safe for a programmer to modify or delete the arrays `row_indices`,  
 2270 `col_indices`, or `values`.

### 2271 4.2.5.10 Matrix\_setElement: Set a single element in matrix

2272 Set one element of a matrix to a given value.

## 2273 C Syntax

```
2274 // scalar value
2275 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2276                               <type>         val,
2277                               GrB_Index        row_index,
2278                               GrB_Index        col_index);
2279
2280 // GraphBLAS scalar
2281 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2282                               const GrB_Scalar s,
2283                               GrB_Index        row_index,
2284                               GrB_Index        col_index);
```



## 2285 Parameters

2286           **C** (INOUT) An existing GraphBLAS matrix for which an element is to be assigned.

2287           **val** or **s** (IN) Scalar to assign. Its domain (type) must be compatible with the domain of

2288           **C**.

2289           **row\_index** (IN) Row index of element to be assigned

2290           **col\_index** (IN) Column index of element to be assigned

## 2291 Return Values

2292           **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-

2293           blocking mode, this indicates that the compatibility tests on in-

2294           dex/dimensions and domains for the input arguments passed suc-

2295           cessfully. Either way, the output matrix **C** is ready to be used in

2296           the next method of the sequence.

2297           **GrB\_PANIC** Unknown internal error.

2298           **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque

2299           GraphBLAS objects (input or output) is in an invalid state caused

2300           by a previous execution error. Call **GrB\_error()** to access any error

2301           messages generated by the implementation.

2302           **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2303           **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, or GraphBLAS scalar, **s**, has not been

2304           initialized by a call to a respective constructor.

2305           **GrB\_INVALID\_INDEX** **row\_index** or **col\_index** is outside the allowable range (i.e., not less

2306           than **nrows(C)** or **ncols(C)**, respectively).

2307           **GrB\_DOMAIN\_MISMATCH** The domains of the matrix and the scalar are incompatible.

## 2308 Description

2309 First, the scalar and output matrix are tested for domain compatibility as follows: **D(val)** or

2310 **D(s)** must be compatible with **D(C)**. Two domains are compatible with each other if values from

2311 one domain can be cast to values in the other domain as per the rules of the C language. In

2312 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-

2313 defined type is only compatible with itself. If any compatibility rule above is violated, execution of

2314 **GrB\_Matrix\_setElement** ends and the domain mismatch error listed above is returned.

2315 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2316 \quad & 0 \leq \text{row\_index} < \text{nrows}(\mathbf{C}), \\ & 0 \leq \text{col\_index} < \text{ncols}(\mathbf{C}) \end{aligned}$$

2317 If either of these conditions is violated, execution of `GrB_Matrix_setElement` ends and the invalid  
 2318 index error listed above is returned.

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

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

2319 In the case of a transparent scalar or if  $\mathbf{L}(s)$  is not empty, then a value will be stored at the  
 2320 specified location in  $C$ , overwriting any value that may have been stored there before. In the case  
 2321 of a GraphBLAS scalar and if  $\mathbf{L}(s)$  is empty, then any value stored at the specified location in  $C$   
 2322 will be removed.

2323 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents  
 2324 of  $C$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with  
 2325 return value `GrB_SUCCESS` and the new content of vector  $C$  is as defined above but may not be  
 2326 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 2327 **4.2.5.11 Matrix\_removeElement: Remove an element from a matrix**

2328 Remove (annihilate) one stored element from a matrix.

#### 2329 **C Syntax**

```
2330      GrB_Info GrB_Matrix_removeElement(GrB_Matrix  C,
2331                                         GrB_Index   row_index,
2332                                         GrB_Index   col_index);
```

#### 2333 **Parameters**

2334 `C` (INOUT) An existing GraphBLAS matrix from which an element is to be removed.

2335 `row_index` (IN) Row index of element to be removed

2336 `col_index` (IN) Column index of element to be removed

#### 2337 **Return Values**

2338 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
 2339 blocking mode, this indicates that the compatibility tests on in-  
 2340 dex/dimensions and domains for the input arguments passed suc-  
 2341 cessfully. Either way, the output matrix  $C$  is ready to be used in  
 2342 the next method of the sequence.

2343 `GrB_PANIC` Unknown internal error.

2344       GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 2345       GraphBLAS objects (input or output) is in an invalid state caused  
 2346       by a previous execution error. Call GrB\_error() to access any error  
 2347       messages generated by the implementation.

2348       GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2349 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, C, has not been initialized by a call to  
 2350       any matrix constructor.

2351       GrB\_INVALID\_INDEX row\_index or col\_index is outside the allowable range (i.e., not less  
 2352       than **nrows(C)** or **ncols(C)**, respectively).

## 2353 Description

2354 First, both index parameters are checked for valid values where following conditions must hold:

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

2356 If either of these conditions is violated, execution of GrB\_Matrix\_removeElement ends and the  
 2357 invalid index error listed above is returned.

2358 We are now ready to carry out the removal of a value that may be stored at the location specified by  
 2359 (row\_index, col\_index). If a value does not exist at the specified location in C, no error is reported  
 2360 and the operation has no effect on the state of C. In either case, the following will be true on return  
 2361 from this method: (row\_index, col\_index)  $\notin$  ind(C)

2362 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new contents  
 2363 of C is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method exits with  
 2364 return value GrB\_SUCCESS and the new content of vector C is as defined above but may not be  
 2365 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 2366 4.2.5.12 Matrix\_extractElement: Extract a single element from a matrix

2367 Extract one element of a matrix into a scalar.

## 2368 C Syntax

```

2369      // scalar value
2370      GrB_Info GrB_Matrix_extractElement(<type>          *val,
2371                                         const GrB_Matrix A,
2372                                         GrB_Index      row_index,
2373                                         GrB_Index      col_index);
2374
2375      // GraphBLAS scalar
  
```

```

2376         GrB_Info GrB_Matrix_extractElement(GrB_Scalar      s,
2377                                           const GrB_Matrix A,
2378                                           GrB_Index      row_index,
2379                                           GrB_Index      col_index);
2380

```

## 2381 Parameters

2382     **val or s (INOUT)** An existing scalar whose domain is compatible with the domain of matrix  
2383     **A.** On successful return, this scalar holds the result of the extract. Any previous  
2384     value stored in **val** or **s** is overwritten.

2385     **A (IN)** The GraphBLAS matrix from which an element is extracted.

2386     **row\_index (IN)** The row index of location in **A** to extract.

2387     **col\_index (IN)** The column index of location in **A** to extract.

## 2388 Return Values

2389     **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2390     cessfully. This indicates that the compatibility tests on dimensions  
2391     and domains for the input arguments passed successfully, and the  
2392     output scalar, **val** or **s**, has been computed and is ready to be used  
2393     in the next method of the sequence.

2394     **GrB\_NO\_VALUE** When using the transparent scalar, **val**, this is returned when there  
2395     is no stored value at specified location.

2396     **GrB\_PANIC** Unknown internal error.

2397     **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2398     GraphBLAS objects (input or output) is in an invalid state caused  
2399     by a previous execution error. Call **GrB\_error()** to access any error  
2400     messages generated by the implementation.

2401     **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2402     **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, or scalar, **s**, has not been initialized by  
2403     a call to a corresponding constructor.

2404     **GrB\_NULL\_POINTER** **val** pointer is **NULL**.

2405     **GrB\_INVALID\_INDEX** **row\_index** or **col\_index** is outside the allowable range (i.e. less than  
2406     zero or greater than or equal to **nrows(A)** or **ncols(A)**, respec-  
2407     tively).

2408     **GrB\_DOMAIN\_MISMATCH** The domains of the matrix and scalar are incompatible.

## 2409 Description

2410 First, the scalar and input matrix are tested for domain compatibility as follows:  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(\mathbf{s})$   
 2411 must be compatible with  $\mathbf{D}(\mathbf{A})$ . Two domains are compatible with each other if values from  
 2412 one domain can be cast to values in the other domain as per the rules of the C language. In  
 2413 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
 2414 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
 2415 `GrB_Matrix_extractElement` ends and the domain mismatch error listed above is returned.

2416 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2417 \quad & 0 \leq \text{row\_index} < \mathbf{nrows}(\mathbf{A}), \\ & 0 \leq \text{col\_index} < \mathbf{ncols}(\mathbf{A}) \end{aligned}$$

2418 If either condition is violated, execution of `GrB_Matrix_extractElement` ends and the invalid index  
 2419 error listed above is returned.

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

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

2420 If  $(\text{row\_index}, \text{col\_index}) \in \mathbf{ind}(\mathbf{A})$ , then the corresponding value from  $\mathbf{A}$  is copied into  $\mathbf{s}$  or  $\text{val}$   
 2421 with casting as necessary. If  $(\text{row\_index}, \text{col\_index}) \notin \mathbf{ind}(\mathbf{A})$ , then one of the follow occurs  
 2422 depending on output scalar type:

- 2423 • The GraphBLAS scalar,  $\mathbf{s}$ , is cleared and `GrB_SUCCESS` is returned.
- 2424 • The non-opaque scalar,  $\text{val}$ , is unchanged, and `GrB_NO_VALUE` is returned.

2425 When using the non-opaque scalar variant ( $\text{val}$ ) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`  
 2426 mode, the new contents of  $\text{val}$  are as defined above if the method exits with return value `GrB_SUCCESS`  
 2427 or `GrB_NO_VALUE`.

2428 When using the GraphBLAS scalar variant ( $\mathbf{s}$ ) with a `GrB_SUCCESS` return value, the method  
 2429 exits and the new contents of  $\mathbf{s}$  is as defined above and fully computed in `GrB_BLOCKING` mode.  
 2430 In `GrB_NONBLOCKING` mode, the new contents of  $\mathbf{s}$  is as defined above but may not be fully  
 2431 computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 2432 4.2.5.13 Matrix\_extractTuples: Extract tuples from a matrix

2433 Extract the contents of a GraphBLAS matrix into non-opaque data structures.

## 2434 C Syntax

```
2435      GrB_Info GrB_Matrix_extractTuples(GrB_Index      *row_indices,
2436                                     GrB_Index      *col_indices,
```

```

2437                                     <type>          *values,
2438                                     GrB_Index         *n,
2439                                     const GrB_Matrix   A);

```

## 2440 Parameters

2441     **row\_indices** (OUT) Pointer to an array of row indices that is large enough to hold all of the  
2442     row indices.

2443     **col\_indices** (OUT) Pointer to an array of column indices that is large enough to hold all of the  
2444     column indices.

2445     **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of  
2446     the stored values whose type is compatible with  $\mathbf{D}(\mathbf{A})$ .

2447     **n** (INOUT) Pointer to a value indicating (in input) the number of elements the **values**,  
2448     **row\_indices**, and **col\_indices** arrays can hold. Upon return, it will contain the  
2449     number of values written to the arrays.

2450     **A** (IN) An existing GraphBLAS matrix.

## 2451 Return Values

2452     **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2453     cessfully. This indicates that the compatibility tests on the input  
2454     argument passed successfully, and the output arrays, **indices** and  
2455     **values**, have been computed.

2456     **GrB\_PANIC** Unknown internal error.

2457     **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2458     GraphBLAS objects (input or output) is in an invalid state caused  
2459     by a previous execution error. Call **GrB\_error()** to access any error  
2460     messages generated by the implementation.

2461     **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2462     **GrB\_INSUFFICIENT\_SPACE** Not enough space in **row\_indices**, **col\_indices**, and **values** (as indi-  
2463     cated by the **n** parameter) to hold all of the tuples that will be  
2464     extracted.

2465     **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to  
2466     any matrix constructor.

2467     **GrB\_NULL\_POINTER** **row\_indices**, **col\_indices**, **values** or **n** pointer is NULL.

2468     **GrB\_DOMAIN\_MISMATCH** The domains of the **A** matrix and **values** array are incompatible  
2469     with one another.

## 2470 Description

2471 This method will extract all the tuples from the GraphBLAS matrix **A**. The values associated with  
2472 those tuples are placed in the **values** array, the column indices are placed in the **col\_indices** array,  
2473 and the row indices are placed in the **row\_indices** array. These output arrays are pre-allocated by  
2474 the user before calling this function such that each output array has enough space to hold at least  
2475 **GrB\_Matrix\_nvals(A)** elements.

2476 Upon return of this function, a pair of  $\{\text{row\_indices}[k], \text{col\_indices}[k]\}$  are unique for every valid  
2477  $k$ , but they are not required to be sorted in any particular order. Each tuple  $(i, j, A_{ij})$  in **A** is  
2478 unzipped and copied into a distinct  $k$ th location in output vectors:

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

2479 where  $0 \leq k < \text{GrB\_Matrix\_nvals}(v)$ . No gaps in output vectors are allowed; that is, if **row\_indices**[ $k$ ],  
2480 **col\_indices**[ $k$ ] and **values**[ $k$ ] exist upon return, so does **row\_indices**[ $j$ ], **col\_indices**[ $j$ ] and **values**[ $j$ ] for  
2481 all  $j$  such that  $0 \leq j < k$ .

2482 Note that if the value in **n** on input is less than the number of values contained in the matrix **A**,  
2483 then a **GrB\_INSUFFICIENT\_SPACE** error is returned since it is undefined which subset of values  
2484 would be extracted.

2485 In both **GrB\_BLOCKING** mode **GrB\_NONBLOCKING** mode if the method exits with return value  
2486 **GrB\_SUCCESS**, the new contents of the arrays **row\_indices**, **col\_indices** and **values** are as defined  
2487 above.

2488 **4.2.5.14 Matrix\_exportHint: Provide a hint as to which storage format might be most**  
2489 **efficient for exporting a matrix**

## 2490 C Syntax

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

## 2491 Parameters

2492 **hint** (OUT) Pointer to a value of type **GrB\_Format**.

2493 **A** (IN) A GraphBLAS matrix object.

## 2494 Return Values

2495 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2496 cessfully and the value of **hint** has been set.

2497 **GrB\_PANIC** Unknown internal error.

2498           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
 2499                                   opaque GraphBLAS objects (input or output) is in an invalid  
 2500                                   state caused by a previous execution error. Call GrB\_error() to  
 2501                                   access any error messages generated by the implementation.

2502           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2503           GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
 2504                                   any matrix constructor.

2505           GrB\_NULL\_POINTER hint is NULL.

2506           GrB\_NO\_VALUE If the implementation does not have a preferred format, it may  
 2507                                   return the value GrB\_NO\_VALUE.

## 2508 Description

2509   Given a GraphBLAS matrix A, provide a hint as to which format might be most efficient for  
 2510   exporting the matrix A. GraphBLAS implementations might return the current storage format of  
 2511   the matrix, or the format to which it could most efficiently be exported. However, implementations  
 2512   are free to return any value for format defined in Section 3.5.3.1. Note that an implementation is  
 2513   free to refuse to provide a format hint, returning GrB\_NO\_VALUE.

### 2514 4.2.5.15 Matrix\_exportSize: Return the array sizes necessary to export a GraphBLAS 2515 matrix object

## 2516 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);

```

## 2517 Parameters

2518   n\_indptr (OUT) Pointer to a value of type GrB\_Index.

2519   n\_indices (OUT) Pointer to a value of type GrB\_Index.

2520   n\_values (OUT) Pointer to a value of type GrB\_Index.

2521   format (IN) a value indicating the format in which the matrix will be exported, as defined  
 2522           in Section 3.5.3.1.

2523   A (IN) A GraphBLAS matrix object.



## 2524 Return Values

2525                   GrB\_SUCCESS In blocking mode or non-blocking mode, the operation com-  
2526                   pleted successfully. This indicates that the API checks for the  
2527                   input arguments passed successfully, and the number of elements  
2528                   necessary for the export buffers have been written to `n_indptr`,  
2529                   `n_indices`, and `n_values`, respectively.

2530                   GrB\_PANIC Unknown internal error.

2531                   GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
2532                   opaque GraphBLAS objects (input or output) is in an invalid  
2533                   state caused by a previous execution error. Call `GrB_error()` to  
2534                   access any error messages generated by the implementation.

2535                   GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2536                   GrB\_UNINITIALIZED\_OBJECT The GraphBLAS Matrix, `A`, has not been initialized by a call to  
2537                   any matrix constructor.

2538                   GrB\_NULL\_POINTER `n_indptr`, `n_indices`, or `n_values` is NULL.

## 2539 Description

2540   Given a matrix `A`, returns the required capacities of arrays `values`, `indptr`, and `indices` necessary to  
2541   export the matrix in the format specified by `format`. The output values `n_values`, `n_indptr`, and  
2542   `indices` will contain the corresponding sizes of the arrays (in number of elements) that must be  
2543   allocated to hold the exported matrix. The argument `format` can be chosen arbitrarily by the user  
2544   as one of the values defined in Section 3.5.3.1.

### 2545 4.2.5.16 Matrix\_export: Export a GraphBLAS matrix to a pre-defined format

## 2546 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);
```

## 2547 Parameters

- 2548        **indptr** (INOUT) Pointer to an array that will hold row or column offsets, or row in-  
2549        dices, depending on the value of **format**. It must be large enough to hold at  
2550        least **n\_indptr** elements of type **GrB\_Index**, where **n\_indices** was returned from  
2551        **GrB\_Matrix\_exportSize()** method.
- 2552        **indices** (INOUT) Pointer to an array that will hold row or column indices of the elements  
2553        in **values**, depending on the value of **format**. It must be large enough to hold at  
2554        least **n\_indices** elements of type **GrB\_Index**, where **n\_indices** was returned from  
2555        **GrB\_Matrix\_exportSize()** method.
- 2556        **values** (INOUT) Pointer to an array that will hold stored values. The type of ele-  
2557        ment must match the type of the values stored in **A**. It must be large enough  
2558        to hold at least **n\_values** elements of that type, where **n\_values** was returned from  
2559        **GrB\_Matrix\_exportSize**.
- 2560        **n\_indptr** (INOUT) Pointer to a value indicating (on input) the number of elements the **indptr**  
2561        array can hold. Upon return, it will contain the number of elements written to the  
2562        array.
- 2563        **n\_indices** (INOUT) Pointer to a value indicating (on input) the number of elements the **indices**  
2564        array can hold. Upon return, it will contain the number of elements written to the  
2565        array.
- 2566        **n\_values** (INOUT) Pointer to a value indicating (on input) the number of elements the **values**  
2567        array can hold. Upon return, it will contain the number of elements written to the  
2568        array.
- 2569        **format** (IN) a value indicating the format in which the matrix will be exported, as defined  
2570        in Section 3.5.3.1.
- 2571        **A** (IN) A GraphBLAS matrix object.

## 2572 Return Values

- 2573        **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2574        cessfully. This indicates that the compatibility tests on the input  
2575        argument passed successfully, and the output arrays, **indptr**, **in-**  
2576        **dices** and **values**, have been computed.
- 2577        **GrB\_PANIC** Unknown internal error.
- 2578        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
2579        opaque GraphBLAS objects (input or output) is in an invalid  
2580        state caused by a previous execution error. Call **GrB\_error()** to  
2581        access any error messages generated by the implementation.
- 2582        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2583       GrB\_INSUFFICIENT\_SPACE Not enough space in `indptr`, `indices`, and/or `values` (as indicated  
2584       by the corresponding `n_*` parameter) to hold all of the corre-  
2585       sponding elements that will be extacted.

2586       GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to  
2587       any matrix constructor.

2588       GrB\_NULL\_POINTER `indptr`, `indices`, `values` `n_indptr`, `n_indices`, `n_values` pointer is  
2589       NULL.

2590       GrB\_DOMAIN\_MISMATCH The domain of `A` does not match with the type of `values`.

## 2591 Description

2592 Given a matrix `A`, this method exports the contents of the matrix into one of the pre-defined  
2593 `GrB_Format` formats from Section 3.5.3.1. The user-allocated arrays pointed to by `indptr`, `indices`,  
2594 and `values` must be at least large enough to hold the corresponding number of elements returned  
2595 by calling `GrB_Matrix_exportSize`. The value of `format` can be chosen arbitrarily, but a call to  
2596 `GrB_Matrix_exportHint` may suggest a format that results in the most efficient export. Details  
2597 of the contents of `indptr`, `indices`, and `values` corresponding to each supported format is given in  
2598 Appendix B.

### 2599 4.2.5.17 Matrix\_import: Import a matrix into a GraphBLAS object

## 2600 C Syntax

```

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

```

## 2601 Parameters

2602       `A` (INOUT) On a successful return, contains a handle to the newly created Graph-  
2603       BLAS matrix.

2604       `d` (IN) The type corresponding to the domain of the matrix being created. Can be  
2605       one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
2606       GraphBLAS type.

2607        **nrows** (IN) Integer value holding the number of rows in the matrix.

2608        **ncols** (IN) Integer value holding the number of columns in the matrix.

2609        **indptr** (IN) Pointer to an array of row or column offsets, or row indices, depending on the  
2610                value of **format**.

2611        **indices** (IN) Pointer to an array row or column indices of the elements in **values**, depending  
2612                on the value of **format**.

2613        **values** (IN) Pointer to an array of values. Type must match the type of **d**.

2614        **n\_indptr** (IN) Integer value holding the number of elements in the array pointed to by **indptr**.

2615        **n\_indices** (IN) Integer value holding the number of elements in the array pointed to by **indices**.

2616        **n\_values** (IN) Integer value holding the number of elements in the array pointed to by **values**.

2617        **format** (IN) a value indicating the format of the matrix being imported, as defined in  
2618                Section 3.5.3.1.

## 2619    **Return Values**

2620        **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2621                blocking mode, this indicates that the API checks for the input  
2622                arguments passed successfully and the input arrays have been  
2623                consumed. Either way, output matrix **A** is ready to be used in  
2624                the next method of the sequence.

2625        **GrB\_PANIC** Unknown internal error.

2626        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2627        **GrB\_UNINITIALIZED\_OBJECT** The **GrB\_Type** object has not been initialized by a call to **GrB\_Type\_new**  
2628                (needed for user-defined types).

2629        **GrB\_NULL\_POINTER** **A**, **indptr**, **indices** or **values** pointer is **NULL**.

2630        **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **indptr** or **indices** is outside the allowed range for indices  
2631                in **A** and or the size of **values**, **n\_values**, depending on the value  
2632                of **format**.

2633        **GrB\_INVALID\_VALUE** **nrows** or **ncols** is zero or outside the range of the type **GrB\_Index**.

2634        **GrB\_DOMAIN\_MISMATCH** The domain given in parameter **d** does not match the element  
2635                type of **values**.

## 2636 Description

2637 Creates a new matrix **A** of domain **D**(d) and dimension **nrows**  $\times$  **ncols**. The new GraphBLAS  
2638 matrix will be filled with the contents of the matrix pointed to by **indptr**, and **indices**, and **values**.  
2639 The method returns a handle to the new matrix in **A**. The structure of the data being imported is  
2640 defined by **format**, which must be equal to one of the values defined in Section 3.5.3.1. Details of  
2641 the contents of **indptr**, **indices** and **values** for each supported format is given in Appendix B.

2642 It is not an error to call this method more than once on the same output matrix; however, the  
2643 handle to the previously created object will be overwritten.

## 2644 4.2.5.18 Matrix\_serializeSize: Compute the serialize buffer size

2645 Compute the buffer size (in bytes) necessary to serialize a GrB\_Matrix using GrB\_Matrix\_serialize.

## 2646 C Syntax

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

## 2647 Parameters

2648 **size** (OUT) Pointer to GrB\_Index value where size in bytes of serialized object will be  
2649 written.

2650 **A** (IN) A GraphBLAS matrix object.

## 2651 Return Values

2652 **GrB\_SUCCESS** The operation completed successfully and the value pointed to  
2653 by **\*size** has been computed and is ready to use.

2654 **GrB\_PANIC** Unknown internal error.

2655 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2656 **GrB\_NULL\_POINTER** **size** is NULL.

## 2657 Description

2658 Returns the size in bytes of the data buffer necessary to serialize the GraphBLAS matrix object **A**.  
2659 Users may then allocate a buffer of **size** bytes to pass as a parameter to GrB\_Matrix\_serialize.

2660 **4.2.5.19 Matrix\_serialize: Serialize a GraphBLAS matrix.**

2661 Serialize a GraphBLAS Matrix object into an opaque stream of bytes.

2662 **C Syntax**

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

2663 **Parameters**

2664 **serialized\_data** (INOUT) Pointer to the preallocated buffer where the serialized matrix will be  
2665 written.

2666 **serialized\_size** (INOUT) On input, the size in bytes of the buffer pointed to by **serialized\_data**.  
2667 On output, the number of bytes written to **serialized\_data**.

2668 **A** (IN) A GraphBLAS matrix object.

2669 **Return Values**

2670 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2671 cessfully. This indicates that the compatibility tests on the in-  
2672 put argument passed successfully, and the output buffer **serial-  
2673 ized\_data** and **serialized\_size**, have been computed and are ready  
2674 to use.

2675 **GrB\_PANIC** Unknown internal error.

2676 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
2677 opaque GraphBLAS objects (input or output) is in an invalid  
2678 state caused by a previous execution error. Call **GrB\_error()** to  
2679 access any error messages generated by the implementation.

2680 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2681 **GrB\_NULL\_POINTER** **serialized\_data** or **serialize\_size** is NULL.

2682 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to  
2683 any matrix constructor.

2684 **GrB\_INSUFFICIENT\_SPACE** The size of the buffer **serialized\_data** (provided as an input **seri-  
2685 alized\_size**) was not large enough.

## 2686 Description

2687 Serializes a GraphBLAS matrix object to an opaque buffer. To guarantee successful execution,  
2688 the size of the buffer pointed to by `serialized_data`, provided as an input by `serialized_size`, must  
2689 be of at least the number of bytes returned from `GrB_Matrix_serializeSize`. The actual size of the  
2690 serialized matrix written to `serialized_data` is provided upon completion as an output written to  
2691 `serialized_size`.

2692 The contents of the serialized buffer are implementation defined. Thus, a serialized matrix created  
2693 with one library implementation is not necessarily valid for deserialization with another implemen-  
2694 tation.

### 2695 4.2.5.20 Matrix\_deserialize: Deserialize a GraphBLAS matrix.

2696 Construct a new GraphBLAS matrix from a serialized object.

## 2697 C Syntax

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

## 2698 Parameters

2699 A (INOUT) On a successful return, contains a handle to the newly created Graph-  
2700 BLAS matrix.

2701 d (IN) the type of the matrix that was serialized in `serialized_data`.

2702 `serialized_data` (IN) a pointer to a serialized GraphBLAS matrix created with `GrB_Matrix_serialize`.

2703 `serialized_size` (IN) the size of the buffer pointed to by `serialized_data` in bytes.

## 2704 Return Values

2705 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2706 blocking mode, this indicates that the API checks for the input  
2707 arguments passed successfully. Either way, output matrix A is  
2708 ready to be used in the next method of the sequence.

2709 GrB\_PANIC Unknown internal error.

2710 GrB\_INVALID\_OBJECT This is returned if `serialized_data` is invalid or corrupted.

2711 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2712 GrB\_UNINITIALIZED\_OBJECT The GrB\_Type object has not been initialized by a call to GrB\_Type\_new  
2713 (needed for user-defined types).

2714 GrB\_NULL\_POINTER serialized\_data or A is NULL.

2715 GrB\_DOMAIN\_MISMATCH The type given in d does not match the type of the matrix  
2716 serialized in serialized\_data.

## 2717 Description

2718 Creates a new matrix **A** using the serialized matrix object pointed to by `serialized_data`. The object  
2719 pointed to by `serialized_data` must have been created using the method `GrB_Matrix_serialize`. The  
2720 domain of the matrix is given as an input in `d`, which must match the domain of the matrix serialized  
2721 in `serialized_data`. Note that for user-defined types, only the size of the type will be checked.

2722 Since the format of a serialized matrix is implementation-defined, it is not guaranteed that a matrix  
2723 serialized in one library implementation can be deserialized by another.

2724 It is not an error to call this method more than once on the same output matrix; however, the  
2725 handle to the previously created object will be overwritten.

## 2726 4.2.6 Descriptor methods

2727 The methods in this section create and set values in descriptors. A descriptor is an opaque Graph-  
2728 BLAS object the values of which are used to modify the behavior of GraphBLAS operations.

### 2729 4.2.6.1 Descriptor\_new: Create new descriptor

2730 Creates a new (empty or default) descriptor.

## 2731 C Syntax

2732 GrB\_Info GrB\_Descriptor\_new(GrB\_Descriptor \*desc);

## 2733 Parameters

2734 desc (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2735 descriptor.

## 2736 Return Value

2737 GrB\_SUCCESS The method completed successfully.

2738 GrB\_PANIC unknown internal error.



2739        GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

2740        GrB\_NULL\_POINTER desc pointer is NULL.

## 2741 **Description**

2742        Creates a new descriptor object and returns a handle to it in desc. A newly created descriptor can  
2743        be populated by calls to Descriptor\_set.

2744        It is not an error to call this method more than once on the same variable; however, the handle to  
2745        the previously created object will be overwritten.

### 2746 **4.2.6.2 Descriptor\_set: Set content of descriptor**

2747        Sets the content for a field for an existing descriptor.

## 2748 **C Syntax**

```
2749        GrB_Info GrB_Descriptor_set(GrB_Descriptor        desc,  
2750                                    GrB_Desc_Field        field,  
2751                                    GrB_Desc_Value        val);
```

## 2752 **Parameters**

2753        desc (IN) An existing GraphBLAS descriptor to be modified.

2754        field (IN) The field being set.

2755        val (IN) New value for the field being set.

## 2756 **Return Values**

2757        GrB\_SUCCESS operation completed successfully.

2758        GrB\_PANIC unknown internal error.

2759        GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

2760        GrB\_UNINITIALIZED\_OBJECT the desc parameter has not been initialized by a call to new.

2761        GrB\_INVALID\_VALUE invalid value set on the field, or invalid field.

## 2762 Description

2763 For a given descriptor, the `GrB_Descriptor_set` method can be called for each field in the descriptor  
2764 to set the value associated with that field. Valid values for the `field` parameter include the following:

2765 `GrB_OUTP` refers to the output parameter (result) of the operation.

2766 `GrB_MASK` refers to the mask parameter of the operation.

2767 `GrB_INP0` refers to the first input parameters of the operation (matrices and vectors).

2768 `GrB_INP1` refers to the second input parameters of the operation (matrices and vectors).

2769 Valid values for the `val` parameter are:

2770 `GrB_STRUCTURE` Use only the structure of the stored values of the corresponding mask  
2771 (`GrB_MASK`) parameter.

2772 `GrB_COMP` Use the complement of the corresponding mask (`GrB_MASK`) param-  
2773 eter. When combined with `GrB_STRUCTURE`, the complement of the  
2774 structure of the mask is used without evaluating the values stored.

2775 `GrB_TRAN` Use the transpose of the corresponding matrix parameter (valid for input  
2776 matrix parameters only).

2777 `GrB_REPLACE` When assigning the masked values to the output matrix or vector, clear  
2778 the matrix first (or clear the non-masked entries). The default behavior  
2779 is to leave non-masked locations unchanged. Valid for the `GrB_OUTP`  
2780 parameter only.

2781 Descriptor values can only be set, and once set, cannot be cleared. As, in the case of `GrB_MASK`,  
2782 multiple values can be set and all will apply (for example, both `GrB_COMP` and `GrB_STRUCTURE`).  
2783 A value for a given field may be set multiple times but will have no additional effect. Fields that  
2784 have no values set result in their default behavior, as defined in Section 3.7.

## 2785 4.2.7 free: Destroy an object and release its resources

2786 Destroys a previously created GraphBLAS object and releases any resources associated with the  
2787 object.

## 2788 C Syntax

2789 `GrB_Info GrB_free(<GrB_Object> *obj);`

## 2790 Parameters

2791       obj (INOUT) An existing GraphBLAS object to be destroyed. The object must have  
2792       been created by an explicit call to a GraphBLAS constructor. It can be any of the  
2793       opaque GraphBLAS objects such as matrix, vector, descriptor, semiring, monoid,  
2794       binary op, unary op, or type. On successful completion of GrB\_free, obj behaves  
2795       as an uninitialized object.

## 2796 Return Values

2797       GrB\_SUCCESS operation completed successfully

2798       GrB\_PANIC unknown internal error. If this return value is encountered when  
2799       in nonblocking mode, the error responsible for the panic condition  
2800       could be from any method involved in the computation of the input  
2801       object. The GrB\_error() method should be called for additional  
2802       information.

## 2803 Description

2804       GraphBLAS objects consume memory and other resources managed by the GraphBLAS runtime  
2805       system. A call to GrB\_free frees those resources so they are available for use by other GraphBLAS  
2806       objects.

2807       The parameter passed into GrB\_free is a handle referencing a GraphBLAS opaque object of a data  
2808       type from table 2.1. The object must have been created by an explicit call to a GraphBLAS con-  
2809       structor. The behavior of a program that calls GrB\_free on a pre-defined object is implementation  
2810       defined.

2811       After the GrB\_free method returns, the object referenced by the input handle is destroyed and the  
2812       handle has the value GrB\_INVALID\_HANDLE. The handle can be used in subsequent GraphBLAS  
2813       methods but only after the handle has been reinitialized with a call the the appropriate \_new or  
2814       \_dup method.

2815       Note that unlike other GraphBLAS methods, calling GrB\_free with an object with an invalid handle  
2816       is legal. The system may attempt to free resources that might be associated with that object, if  
2817       possible, and return normally.

2818       When using GrB\_free it is possible to create a dangling reference to an object. This would occur  
2819       when a handle is assigned to a second variable of the same opaque type. This creates two handles  
2820       that reference the same object. If GrB\_free is called with one of the variables, the object is destroyed  
2821       and the handle associated with the other variable no longer references a valid object. This is not an  
2822       error condition that the implementation of the GraphBLAS API can be expected to catch, hence  
2823       programmers must take care to prevent this situation from occurring.

#### 2824 4.2.8 wait: Return once an object is either *complete* or *materialized*

2825 Wait until method calls in a sequence put an object into a state of *completion* or *materialization*.

#### 2826 C Syntax

```
2827 GrB_Info GrB_wait(GrB_Object obj, GrB_WaitMode mode);
```

#### 2828 Parameters

2829 **obj** (INOUT) An existing GraphBLAS object. The object must have been created by an  
2830 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS  
2831 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,  
2832 or type. On successful return of `GrB_wait`, the **obj** can be safely read from another  
2833 thread (completion) or all computing to produce **obj** by all GraphBLAS operations  
2834 in its sequence have finished (materialization).

2835 **mode** (IN) Set's the mode for `GrB_wait` for whether it is waiting for **obj** to be in the  
2836 state of *completion* or *materialization*. Acceptable values are `GrB_COMPLETE` or  
2837 `GrB_MATERIALIZE`.

#### 2838 Return values

2839 `GrB_SUCCESS` operation completed successfully.

2840 `GrB_INDEX_OUT_OF_BOUNDS` an index out-of-bounds execution error happened during com-  
2841 pletion of pending operations.

2842 `GrB_OUT_OF_MEMORY` and out-of-memory execution error happened during completion  
2843 of pending operations.

2844 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`,  
2845 or other constructor, method.

2846 `GrB_PANIC` unknown internal error.

2847 `GrB_INVALID_VALUE` method called with a `GrB_WaitMode` other than `GrB_COMPLETE`  
2848 `GrB_MATERIALIZE`.

#### 2849 Description

2850 On successful return from `GrB_wait()`, the input object, **obj** is in one of two states depending on  
2851 the mode of `GrB_wait`:

- 2852 • *complete*: `obj` can be used in a happens-before relation, so in a properly synchronized program  
2853 it can be safely used as an IN or INOUT parameter in a GraphBLAS method call from another  
2854 thread. This result occurs when the mode parameter is set to `GrB_COMPLETE`.
- 2855 • *materialized*: `obj` is *complete*, but in addition, no further computing will be carried out on  
2856 behalf of `obj` and error information is available. This result occurs when the mode parameter  
2857 is set to `GrB_MATERIALIZE`.

2858 Since in blocking mode OUT or INOUT parameters to any method call are materialized upon return,  
2859 `GrB_wait(obj,mode)` has no effect when called in blocking mode.

2860 In non-blocking mode, the status of any pending method calls, other than those associated with pro-  
2861 ducing the *complete* or *materialized* state of `obj`, are not impacted by the call to `GrB_wait(obj,mode)`.  
2862 Methods in the sequence for `obj`, however, most likely would be impacted by a call to `GrB_wait(obj,mode)`;  
2863 especially in the case of the *materialized* mode for which any computing on behalf of `obj` must be  
2864 finished prior to the return from `GrB_wait(obj,mode)`.

#### 2865 4.2.9 error: Retrieve an error string

2866 Retrieve an error-message about any errors encountered during the processing associated with an  
2867 object.

### 2868 C Syntax

```
2869         GrB_Info GrB_error(const char          **error,
2870                           const GrB_Object      obj);
```

#### 2871 Parameters

2872 `error` (OUT) A pointer to a null-terminated string. The contents of the string are im-  
2873 plementation defined.

2874 `obj` (IN) An existing GraphBLAS object. The object must have been created by an  
2875 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS  
2876 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,  
2877 or type.

#### 2878 Return value

2879 `GrB_SUCCESS` operation completed successfully.

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.

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

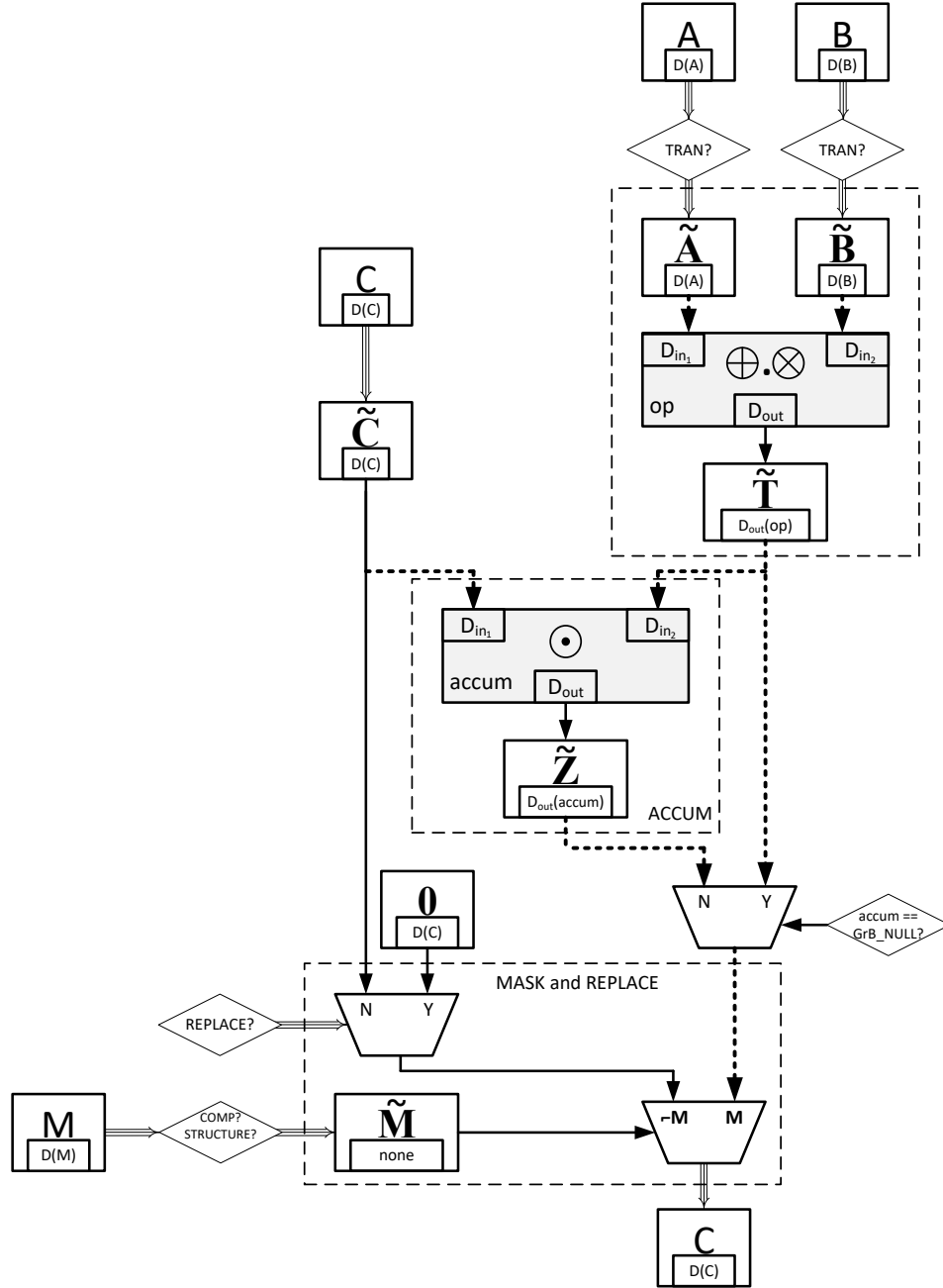


Figure 4.1: Flowchart for the GraphBLAS operations. Although shown specifically for the mxm operation, many elements are common to all operations: such as the “ACCUM” and “MASK and REPLACE” blocks. The triple arrows ( $\Rightarrow$ ) denote where “as if copy” takes place (including both collections and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

2952 To support the detection of as many cases of uninitialized objects as possible, it is strongly rec-  
 2953 ommended to initialize all GraphBLAS objects to the predefined value `GrB_INVALID_HANDLE` at  
 2954 the point of their declaration, as shown in the following examples:

```
2955         GrB_Type          type = GrB_INVALID_HANDLE;
2956         GrB_Semiring      semiring = GrB_INVALID_HANDLE;
2957         GrB_Matrix        matrix = GrB_INVALID_HANDLE;
```

## 2958 Compliance

2959 We follow a *prescriptive* approach to the definition of the semantics of GraphBLAS operations.  
 2960 That is, for each operation we give a recipe for producing its outcome. Any implementation that  
 2961 produces the same outcome, and follows the GraphBLAS execution model (Section 2.5) and error  
 2962 model (Section 2.6) is a conforming implementation.

### 2963 4.3.1 mxm: Matrix-matrix multiply

2964 Multiplies a matrix with another matrix on a semiring. The result is a matrix.

## 2965 C Syntax

```
2966         GrB_Info GrB_mxm(GrB_Matrix          C,
2967                         const GrB_Matrix      Mask,
2968                         const GrB_BinaryOp     accum,
2969                         const GrB_Semiring     op,
2970                         const GrB_Matrix      A,
2971                         const GrB_Matrix      B,
2972                         const GrB_Descriptor   desc);
```

## 2973 Parameters

2974 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
 2975 that may be accumulated with the result of the matrix product. On output, the  
 2976 matrix holds the results of the operation.

2977 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
 2978 stored into the output matrix C. The mask dimensions must match those of the  
 2979 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
 2980 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types  
 2981 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
 2982 dimensions of C), `GrB_NULL` should be specified.

2983 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
 2984 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
 2985 specified.

2986 **op** (IN) The semiring used in the matrix-matrix multiply.

2987 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
 2988 multiplication.

2989 **B** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
 2990 multiplication.

2991 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
 2992 should be specified. Non-default field/value pairs are listed as follows:  
 2993

Param	Field	Value	Description
<b>C</b>	<b>GrB_OUTP</b>	<b>GrB_REPLACE</b>	Output matrix <b>C</b> is cleared (all elements removed) before the result is stored in it.
<b>Mask</b>	<b>GrB_MASK</b>	<b>GrB_STRUCTURE</b>	The write mask is constructed from the structure (pattern of stored values) of the input <b>Mask</b> matrix. The stored values are not examined.
<b>Mask</b>	<b>GrB_MASK</b>	<b>GrB_COMP</b>	Use the complement of <b>Mask</b> .
<b>A</b>	<b>GrB_INP0</b>	<b>GrB_TRAN</b>	Use transpose of <b>A</b> for the operation.
<b>B</b>	<b>GrB_INP1</b>	<b>GrB_TRAN</b>	Use transpose of <b>B</b> for the operation.

## 2995 Return Values

2996 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 2997 blocking mode, this indicates that the compatibility tests on di-  
 2998 mensions and domains for the input arguments passed successfully.  
 2999 Either way, output matrix **C** is ready to be used in the next method  
 3000 of the sequence.

3001 **GrB\_PANIC** Unknown internal error.

3002 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 3003 GraphBLAS objects (input or output) is in an invalid state caused  
 3004 by a previous execution error. Call **GrB\_error()** to access any error  
 3005 messages generated by the implementation.

3006 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

3007 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 3008 a call to **new** (or **Matrix\_dup** for matrix parameters).

3009 **GrB\_DIMENSION\_MISMATCH** Mask and/or matrix dimensions are incompatible.

3010 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
 3011 corresponding domains of the semiring or accumulation operator,  
 3012 or the mask's domain is not compatible with `bool` (in the case where  
 3013 `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3014 Description

3015 GrB\_mxm computes the matrix product  $C = A \oplus . \otimes B$  or, if an optional binary accumulation operator  
 3016  $(\odot)$  is provided,  $C = C \odot (A \oplus . \otimes B)$  (where matrices  $A$  and  $B$  can be optionally transposed).  
 3017 Logically, this operation occurs in three steps:

3018 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 3019 and dimensions are tested for compatibility.

3020 **Compute** The indicated computations are carried out.

3021 **Output** The result is written into the output matrix, possibly under control of a mask.

3022 Up to four argument matrices are used in the GrB\_mxm operation:

- 3023 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3024 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)
- 3025 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3026 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3027 The argument matrices, the semiring, and the accumulation operator (if provided) are tested for  
 3028 domain compatibility as follows:

- 3029 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 3030 must be from one of the pre-defined types of Table 3.2.
- 3031 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the semiring.
- 3032 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the semiring.
- 3033 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the semiring.
- 3034 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 3035 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
 3036 of the accumulation operator.

3037 Two domains are compatible with each other if values from one domain can be cast to values in  
 3038 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are  
 3039 all compatible with each other. A domain from a user-defined type is only compatible with itself.

3040 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the domain mismatch  
 3041 error listed above is returned.

3042 From the argument matrices, the internal matrices and mask used in the computation are formed  
 3043 ( $\leftarrow$  denotes copy):

- 3044 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 3045 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
  - 3046 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 3047  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 3048 (b) If `Mask  $\neq$  GrB_NULL`,
    - 3049 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
 3050  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 3051 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
 3052  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 3053 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 3054 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 3055 4. Matrix  $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}$ .

3056 The internal matrices and masks are checked for dimension compatibility. The following conditions  
 3057 must hold:

- 3058 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 3059 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 3060 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
- 3061 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$ .
- 3062 5.  $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$ .

3063 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the dimension mismatch  
 3064 error listed above is returned.

3065 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3066 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3067 We are now ready to carry out the matrix multiplication and any additional associated operations.  
 3068 We describe this in terms of two intermediate matrices:

- 3069 •  $\tilde{\mathbf{T}}$ : The matrix holding the product of matrices  $\tilde{\mathbf{A}}$  and  $\tilde{\mathbf{B}}$ .
- 3070 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

3071 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

$$3073 \quad T_{ij} = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{B}}(:, j))} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{B}}(k, j)),$$

3074 where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring  $\mathbf{op}$ , respectively.

3075 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 3076 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 3077 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$3078 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3079 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
3080 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 3081 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 3082 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3083 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3084 \quad & \\ 3085 \end{aligned}$$

3086 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

3087 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
3088 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
3089 mask which acts as a “write mask”.

- 3090 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
3091 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$3092 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3093 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
3094 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
3095 mask are unchanged:

$$3096 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3097 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
3098 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
3099 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
3100 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
3101 sequence.

### 3102 4.3.2 vxm: Vector-matrix multiply

3103 Multiplies a (row) vector with a matrix on an semiring. The result is a vector.

#### 3104 C Syntax

```
3105      GrB_Info GrB_vxm(GrB_Vector      w,  
3106                      const GrB_Vector mask,  
3107                      const GrB_BinaryOp accum,  
3108                      const GrB_Semiring op,  
3109                      const GrB_Vector u,  
3110                      const GrB_Matrix A,  
3111                      const GrB_Descriptor desc);
```

#### 3112 Parameters

3113 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3114 that may be accumulated with the result of the vector-matrix product. On output,  
3115 this vector holds the results of the operation.

3116 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3117 stored into the output vector **w**. The mask dimensions must match those of the  
3118 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3119 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3120 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
3121 dimensions of **w**), **GrB\_NULL** should be specified.

3122 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
3123 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
3124 specified.

3125 **op** (IN) Semiring used in the vector-matrix multiply.

3126 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the  
3127 multiplication.

3128 **A** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
3129 multiplication.

3130 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
3131 should be specified. Non-default field/value pairs are listed as follows:  
3132

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.



3160     **Output** The result is written into the output vector, possibly under control of a mask.

3161 Up to four argument vectors or matrices are used in the `GrB_vxm` operation:

- 3162     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3163     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3164     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3165     4.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

3166 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are  
 3167 tested for domain compatibility as follows:

- 3168     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
 3169         must be from one of the pre-defined types of Table 3.2.
- 3170     2.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the semiring.
- 3171     3.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$  of the semiring.
- 3172     4.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the semiring.
- 3173     5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
 3174         of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$   
 3175         of the accumulation operator.

3176 Two domains are compatible with each other if values from one domain can be cast to values in  
 3177 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are  
 3178 all compatible with each other. A domain from a user-defined type is only compatible with itself.  
 3179 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the domain mismatch  
 3180 error listed above is returned.

3181 From the argument vectors and matrices, the internal matrices and mask used in the computation  
 3182 are formed ( $\leftarrow$  denotes copy):

- 3183     1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3184     2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 3185         (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 3186         (b) If `mask`  $\neq$  `GrB_NULL`,
    - 3187             i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 3188             ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}(\mathbf{mask})(i) = \mathbf{true})\} \rangle$ .
  - 3189         (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 3190     3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

3191 4. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP1}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

3192 The internal matrices and masks are checked for shape compatibility. The following conditions  
3193 must hold:

3194 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$ .

3195 2.  $\text{size}(\tilde{\mathbf{w}}) = \text{ncols}(\tilde{\mathbf{A}})$ .

3196 3.  $\text{size}(\tilde{\mathbf{u}}) = \text{nrows}(\tilde{\mathbf{A}})$ .

3197 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the dimension mismatch  
3198 error listed above is returned.

3199 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
3200 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3201 We are now ready to carry out the vector-matrix multiplication and any additional associated  
3202 operations. We describe this in terms of two intermediate vectors:

- 3203 •  $\tilde{\mathbf{t}}$ : The vector holding the product of vector  $\tilde{\mathbf{u}}^T$  and matrix  $\tilde{\mathbf{A}}$ .
- 3204 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3205 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.  
3206 The value of each of its elements is computed by

$$3207 \quad t_j = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{A}}(:, j))} (\tilde{\mathbf{u}}(k) \otimes \tilde{\mathbf{A}}(k, j)),$$

3208 where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring `op`, respectively.

3209 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3210 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 3211 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$3212 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3213 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
3214 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 3215 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 3216 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3217 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3218 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3219 \end{aligned}$$

3220 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3221 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 3222 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3223 mask which acts as a “write mask”.

- 3224 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are  
 3225 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$3226 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3227 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3228 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 3229 mask are unchanged:

$$3230 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3231 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 3232 of vector  $\mathbf{w}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 3233 exits with return value GrB\_SUCCESS and the new content of vector  $\mathbf{w}$  is as defined above but  
 3234 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3235 sequence.

### 3236 4.3.3 mxv: Matrix-vector multiply

3237 Multiplies a matrix by a vector on a semiring. The result is a vector.

## 3238 C Syntax

```
3239      GrB_Info GrB_mxv(GrB_Vector      w,
3240                      const GrB_Vector mask,
3241                      const GrB_BinaryOp accum,
3242                      const GrB_Semiring op,
3243                      const GrB_Matrix A,
3244                      const GrB_Vector u,
3245                      const GrB_Descriptor desc);
```

## 3246 Parameters

3247 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
 3248 that may be accumulated with the result of the matrix-vector product. On output,  
 3249 this vector holds the results of the operation.

3250 **mask** (IN) An optional “write” mask that controls which results from this operation are  
 3251 stored into the output vector  $\mathbf{w}$ . The mask dimensions must match those of the  
 3252 vector  $\mathbf{w}$ . If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain

3253 of the `mask` vector must be of type `bool` or any of the predefined “built-in” types  
 3254 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
 3255 dimensions of `w`), `GrB_NULL` should be specified.

3256 `accum` (IN) An optional binary operator used for accumulating entries into existing `w`  
 3257 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
 3258 specified.

3259 `op` (IN) Semiring used in the vector-matrix multiply.

3260 `A` (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
 3261 multiplication.

3262 `u` (IN) The GraphBLAS vector holding the values for the right-hand vector in the  
 3263 multiplication.

3264 `desc` (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
 3265 should be specified. Non-default field/value pairs are listed as follows:  
 3266

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.

## 3268 Return Values

3269 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
 3270 blocking mode, this indicates that the compatibility tests on di-  
 3271 mensions and domains for the input arguments passed successfully.  
 3272 Either way, output vector `w` is ready to be used in the next method  
 3273 of the sequence.

3274 `GrB_PANIC` Unknown internal error.

3275 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque  
 3276 GraphBLAS objects (input or output) is in an invalid state caused  
 3277 by a previous execution error. Call `GrB_error()` to access any error  
 3278 messages generated by the implementation.

3279 `GrB_OUT_OF_MEMORY` Not enough memory available for the operation.

3280 `GrB_UNINITIALIZED_OBJECT` One or more of the GraphBLAS objects has not been initialized by  
 3281 a call to `new` (or `dup` for matrix or vector parameters).

3282 GrB\_DIMENSION\_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

3283 GrB\_DOMAIN\_MISMATCH The domains of the various vectors/matrices are incompatible with  
3284 the corresponding domains of the semiring or accumulation opera-  
3285 tor, or the mask's domain is not compatible with **bool** (in the case  
3286 where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

## 3287 Description

3288 GrB\_mvx computes the matrix-vector product  $w = A \oplus . \otimes u$ , or, if an optional binary accumulation  
3289 operator ( $\odot$ ) is provided,  $w = w \odot (A \oplus . \otimes u)$  (where matrix  $A$  can be optionally transposed).  
3290 Logically, this operation occurs in three steps:

3291 **Setup** The internal vectors, matrices and mask used in the computation are formed and their  
3292 domains/dimensions are tested for compatibility.

3293 **Compute** The indicated computations are carried out.

3294 **Output** The result is written into the output vector, possibly under control of a mask.

3295 Up to four argument vectors or matrices are used in the GrB\_mvx operation:

- 3296 1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$   
3297 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)  
3298 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$   
3299 4.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

3300 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are  
3301 tested for domain compatibility as follows:

- 3302 1. If **mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{mask})$   
3303 must be from one of the pre-defined types of Table 3.2.  
3304 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the semiring.  
3305 3.  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the semiring.  
3306 4.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the semiring.  
3307 5. If **accum** is not GrB\_NULL, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
3308 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
3309 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

$$t_i = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{A}}(i, :)) \cap \text{ind}(\tilde{\mathbf{u}})} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{u}}(k)),$$

3342 where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring **op**, respectively.

3343 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3344 • If **accum** = **GrB\_NULL**, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 3345 • If **accum** is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$3346 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3347 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 3348 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 3349 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 3350 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3351 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3352 \end{aligned}$$

3353 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

3354 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector **w**,  
 3355 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3356 mask which acts as a “write mask”.

- 3357 • If **desc[GrB\_OUTP].GrB\_REPLACE** is set, then any values in **w** on input to this operation are  
 3358 deleted and the content of the new output vector, **w**, is defined as,

$$3359 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3360 • If **desc[GrB\_OUTP].GrB\_REPLACE** is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3361 copied into the result vector, **w**, and elements of **w** that fall outside the set indicated by the  
 3362 mask are unchanged:

$$3363 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3364 In **GrB\_BLOCKING** mode, the method exits with return value **GrB\_SUCCESS** and the new content  
 3365 of vector **w** is as defined above and fully computed. In **GrB\_NONBLOCKING** mode, the method  
 3366 exits with return value **GrB\_SUCCESS** and the new content of vector **w** is as defined above but  
 3367 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3368 sequence.

#### 3370 4.3.4 eWiseMult: Element-wise multiplication

3371 **Note:** The difference between **eWiseAdd** and **eWiseMult** is not about the element-wise operation  
 3372 but how the index sets are treated. **eWiseAdd** returns an object whose indices are the “union” of  
 3373 the indices of the inputs whereas **eWiseMult** returns an object whose indices are the “intersection”  
 3374 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on  
 3375 the set of values from the resulting index set.

#### 3376 4.3.4.1 eWiseMult: Vector variant

3377 Perform element-wise (general) multiplication on the intersection of elements of two vectors, pro-  
3378 ducing a third vector as result.

#### 3379 C Syntax

```
3380      GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3381                           const GrB_Vector  mask,  
3382                           const GrB_BinaryOp accum,  
3383                           const GrB_Semiring op,  
3384                           const GrB_Vector  u,  
3385                           const GrB_Vector  v,  
3386                           const GrB_Descriptor desc);  
3387  
3388      GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3389                           const GrB_Vector  mask,  
3390                           const GrB_BinaryOp accum,  
3391                           const GrB_Monoid  op,  
3392                           const GrB_Vector  u,  
3393                           const GrB_Vector  v,  
3394                           const GrB_Descriptor desc);  
3395  
3396      GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3397                           const GrB_Vector  mask,  
3398                           const GrB_BinaryOp accum,  
3399                           const GrB_BinaryOp op,  
3400                           const GrB_Vector  u,  
3401                           const GrB_Vector  v,  
3402                           const GrB_Descriptor desc);
```

#### 3403 Parameters

3404 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3405 that may be accumulated with the result of the element-wise operation. On output,  
3406 this vector holds the results of the operation.

3407 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3408 stored into the output vector **w**. The mask dimensions must match those of the  
3409 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3410 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3411 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
3412 dimensions of **w**), **GrB\_NULL** should be specified.

3413 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**



entries. If assignment rather than accumulation is desired, `GrB_NULL` should be specified.

**op** (IN) The semiring, monoid, or binary operator used in the element-wise “product” operation. Depending on which type is passed, the following defines the binary operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

Monoid:  $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$ ; the identity element is ignored.

Semiring:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes(\text{op}) \rangle$ ; the additive monoid is ignored.

**u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the operation.

**v** (IN) The GraphBLAS vector holding the values for the right-hand vector in the operation.

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

Param	Field	Value	Description
<b>w</b>	<code>GrB_OUTP</code>	<code>GrB_REPLACE</code>	Output vector <b>w</b> is cleared (all elements removed) before the result is stored in it.
<b>mask</b>	<code>GrB_MASK</code>	<code>GrB_STRUCTURE</code>	The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined.
<b>mask</b>	<code>GrB_MASK</code>	<code>GrB_COMP</code>	Use the complement of <b>mask</b> .

## Return Values

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

**GrB\_PANIC** Unknown internal error.

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

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

3444 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
 3445 a call to `new` (or `dup` for vector parameters).

3446 GrB\_DIMENSION\_MISMATCH Mask or vector dimensions are incompatible.

3447 GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with the cor-  
 3448 responding domains of the binary operator (`op`) or accumulation  
 3449 operator, or the mask's domain is not compatible with `bool` (in the  
 3450 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3451 Description

3452 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS vectors:  
 3453  $\mathbf{w} = \mathbf{u} \otimes \mathbf{v}$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $\mathbf{w} = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$ .  
 3454 Logically, this operation occurs in three steps:

3455 **Setup** The internal vectors and mask used in the computation are formed and their domains  
 3456 and dimensions are tested for compatibility.

3457 **Compute** The indicated computations are carried out.

3458 **Output** The result is written into the output vector, possibly under control of a mask.

3459 Up to four argument vectors are used in the `GrB_eWiseMult` operation:

- 3460 1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3461 2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3462 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3463 4.  $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

3464 The argument vectors, the “product” operator (`op`), and the accumulation operator (if provided)  
 3465 are tested for domain compatibility as follows:

- 3466 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
 3467 must be from one of the pre-defined types of Table 3.2.
- 3468 2.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$ .
- 3469 3.  $\mathbf{D}(\mathbf{v})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$ .
- 3470 4.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$ .
- 3471 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
 3472 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of  
 3473 the accumulation operator.

3474 Two domains are compatible with each other if values from one domain can be cast to values in  
 3475 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 3476 compatible with each other. A domain from a user-defined type is only compatible with itself. If any  
 3477 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch  
 3478 error listed above is returned.

3479 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
 3480 denotes copy):

- 3481 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3482 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 3483 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .
  - 3484 (b) If `mask  $\neq$  GrB_NULL`,
    - 3485 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
    - 3486 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 3487 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 3488 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 3489 4. Vector  $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$ .

3490 The internal vectors and mask are checked for dimension compatibility. The following conditions  
 3491 must hold:

- 3492 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}}) = \text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{v}})$ .

3493 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension  
 3494 mismatch error listed above is returned.

3495 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3496 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3497 We are now ready to carry out the element-wise “product” and any additional associated operations.  
 3498 We describe this in terms of two intermediate vectors:

- 3499 •  $\tilde{\mathbf{t}}$ : The vector holding the element-wise “product” of  $\tilde{\mathbf{u}}$  and vector  $\tilde{\mathbf{v}}$ .
- 3500 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3501 The intermediate vector  $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, t_i) : \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}}) \neq \emptyset\} \rangle$  is created. The  
 3502 value of each of its elements is computed by:

$$3503 \quad t_i = (\tilde{\mathbf{u}}(i) \otimes \tilde{\mathbf{v}}(i)), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}}))$$

3504 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

3505 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .

3506 • If  $\text{accum}$  is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

3507 
$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3508 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 3509 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

3510 
$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

3511

3512 
$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3513

3514 
$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3515 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3516 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 3517 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3518 mask which acts as a “write mask”.

3519 • If  $\text{desc}[\text{GrB\_OUTP}].\text{GrB\_REPLACE}$  is set, then any values in  $\mathbf{w}$  on input to this operation are  
 3520 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

3521 
$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3522 • If  $\text{desc}[\text{GrB\_OUTP}].\text{GrB\_REPLACE}$  is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3523 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 3524 mask are unchanged:

3525 
$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3526 In **GrB\_BLOCKING** mode, the method exits with return value **GrB\_SUCCESS** and the new content  
 3527 of vector  $\mathbf{w}$  is as defined above and fully computed. In **GrB\_NONBLOCKING** mode, the method  
 3528 exits with return value **GrB\_SUCCESS** and the new content of vector  $\mathbf{w}$  is as defined above but  
 3529 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3530 sequence.

#### 3531 4.3.4.2 eWiseMult: Matrix variant

3532 Perform element-wise (general) multiplication on the intersection of elements of two matrices, pro-  
 3533 ducing a third matrix as result.

## 3534 C Syntax

```

3535     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3536                           const GrB_Matrix Mask,
3537                           const GrB_BinaryOp accum,
3538                           const GrB_Semiring op,
3539                           const GrB_Matrix A,
3540                           const GrB_Matrix B,
3541                           const GrB_Descriptor desc);
3542
3543     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3544                           const GrB_Matrix Mask,
3545                           const GrB_BinaryOp accum,
3546                           const GrB_Monoid op,
3547                           const GrB_Matrix A,
3548                           const GrB_Matrix B,
3549                           const GrB_Descriptor desc);
3550
3551     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3552                           const GrB_Matrix Mask,
3553                           const GrB_BinaryOp accum,
3554                           const GrB_BinaryOp op,
3555                           const GrB_Matrix A,
3556                           const GrB_Matrix B,
3557                           const GrB_Descriptor desc);

```

## 3558 Parameters

3559     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
3560     that may be accumulated with the result of the element-wise operation. On output,  
3561     the matrix holds the results of the operation.

3562     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
3563     stored into the output matrix C. The mask dimensions must match those of the  
3564     matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
3565     of the `Mask` matrix must be of type `bool` or any of the predefined “built-in” types  
3566     in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
3567     dimensions of C), `GrB_NULL` should be specified.

3568     **accum** (IN) An optional binary operator used for accumulating entries into existing C  
3569     entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
3570     specified.

3571     **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”  
3572     operation. Depending on which type is passed, the following defines the binary  
3573     operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

3574  
3575  
3576  
3577  
3578  
  
3579  
3580  
  
3581  
3582  
  
3583  
3584  
3585  
  
  
3586  
  
  
  
  
  
  
  
  
  
3587  
  
3588  
3589  
3590  
3591  
3592  
  
3593  
  
3594  
3595  
3596  
3597  
  
3598  
  
3599  
3600  
  
3601

BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .  
 Monoid:  $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$ ; the identity element is ignored.  
 Semiring:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes(\text{op}) \rangle$ ; the additive monoid is ignored.

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

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

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

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.
B	GrB_INP1	GrB_TRAN	Use transpose of B for the operation.

## Return Values

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

GrB\_PANIC Unknown internal error.

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

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

GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or Matrix\_dup for matrix parameters).

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

3602 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
 3603 corresponding domains of the binary operator ( $\otimes$ ) or accumulation  
 3604 operator, or the mask's domain is not compatible with `bool` (in the  
 3605 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3606 Description

3607 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS matrices:  
 3608  $C = A \otimes B$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot (A \otimes B)$ .  
 3609 Logically, this operation occurs in three steps:

3610 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 3611 and dimensions are tested for compatibility.

3612 **Compute** The indicated computations are carried out.

3613 **Output** The result is written into the output matrix, possibly under control of a mask.

3614 Up to four argument matrices are used in the `GrB_eWiseMult` operation:

- 3615 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3616 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)
- 3617 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3618 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3619 The argument matrices, the “product” operator ( $\otimes$ ), and the accumulation operator (if provided)  
 3620 are tested for domain compatibility as follows:

- 3621 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 3622 must be from one of the pre-defined types of Table 3.2.
- 3623 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\otimes)$ .
- 3624 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\otimes)$ .
- 3625 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\otimes)$ .
- 3626 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 3627 of the accumulation operator and  $\mathbf{D}_{out}(\otimes)$  of  $\otimes$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
 3628 the accumulation operator.

3629 Two domains are compatible with each other if values from one domain can be cast to values in  
 3630 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 3631 compatible with each other. A domain from a user-defined type is only compatible with itself. If any

3632 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch  
 3633 error listed above is returned.

3634 From the argument matrices, the internal matrices and mask used in the computation are formed  
 3635 ( $\leftarrow$  denotes copy):

- 3636 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 3637 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
  - 3638 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 3639  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 3640 (b) If `Mask  $\neq$  GrB_NULL`,
    - 3641 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
 3642  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 3643 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
 3644  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\text{bool})\mathbf{Mask}(i, j) = \text{true}\} \rangle$ .
  - 3645 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 3646 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 3647 4. Matrix  $\tilde{\mathbf{B}} \leftarrow \text{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}$ .

3648 The internal matrices and masks are checked for dimension compatibility. The following conditions  
 3649 must hold:

- 3650 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$ .
- 3651 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$ .

3652 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension  
 3653 mismatch error listed above is returned.

3654 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3655 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3656 We are now ready to carry out the element-wise “product” and any additional associated operations.  
 3657 We describe this in terms of two intermediate matrices:

- 3658 •  $\tilde{\mathbf{T}}$ : The matrix holding the element-wise product of  $\tilde{\mathbf{A}}$  and  $\tilde{\mathbf{B}}$ .
- 3659 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

3660 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\} \rangle$   
 3661 is created. The value of each of its elements is computed by

$$3662 \quad T_{ij} = (\tilde{\mathbf{A}}(i, j) \otimes \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})$$

3663 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:



3664 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .

3665 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$3666 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3667 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 3668 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$3669 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$3670 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$3671 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3674 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3675 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 3676 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 3677 mask which acts as a “write mask”.

3678 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
 3679 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$3680 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3681 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 3682 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 3683 mask are unchanged:

$$3684 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3685 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
 3686 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
 3687 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
 3688 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3689 sequence.

#### 3690 4.3.5 eWiseAdd: Element-wise addition

3691 **Note:** The difference between `eWiseAdd` and `eWiseMult` is not about the element-wise operation  
 3692 but how the index sets are treated. `eWiseAdd` returns an object whose indices are the “union” of  
 3693 the indices of the inputs whereas `eWiseMult` returns an object whose indices are the “intersection”  
 3694 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on  
 3695 the set of values from the resulting index set.

#### 3696 4.3.5.1 eWiseAdd: Vector variant

3697 Perform element-wise (general) addition on the elements of two vectors, producing a third vector  
3698 as result.

#### 3699 C Syntax

```
3700     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3701                          const GrB_Vector mask,  
3702                          const GrB_BinaryOp accum,  
3703                          const GrB_Semiring op,  
3704                          const GrB_Vector u,  
3705                          const GrB_Vector v,  
3706                          const GrB_Descriptor desc);  
3707  
3708     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3709                          const GrB_Vector mask,  
3710                          const GrB_BinaryOp accum,  
3711                          const GrB_Monoid op,  
3712                          const GrB_Vector u,  
3713                          const GrB_Vector v,  
3714                          const GrB_Descriptor desc);  
3715  
3716     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3717                          const GrB_Vector mask,  
3718                          const GrB_BinaryOp accum,  
3719                          const GrB_BinaryOp op,  
3720                          const GrB_Vector u,  
3721                          const GrB_Vector v,  
3722                          const GrB_Descriptor desc);
```

#### 3723 Parameters

3724 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3725 that may be accumulated with the result of the element-wise operation. On output,  
3726 this vector holds the results of the operation.

3727 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3728 stored into the output vector **w**. The mask dimensions must match those of the  
3729 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3730 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3731 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
3732 dimensions of **w**), **GrB\_NULL** should be specified.

3733 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**

entries. If assignment rather than accumulation is desired, `GrB_NULL` should be specified.

**op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum” operation. Depending on which type is passed, the following defines the binary operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$ , used:

BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

Monoid:  $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$ ; the identity element is ignored.

Semiring:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus(\text{op}) \rangle$ ; the multiplicative binary op and additive identity are ignored.

**u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the operation.

**v** (IN) The GraphBLAS vector holding the values for the right-hand vector in the operation.

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

Param	Field	Value	Description
<b>w</b>	<code>GrB_OUTP</code>	<code>GrB_REPLACE</code>	Output vector <b>w</b> is cleared (all elements removed) before the result is stored in it.
<b>mask</b>	<code>GrB_MASK</code>	<code>GrB_STRUCTURE</code>	The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined.
<b>mask</b>	<code>GrB_MASK</code>	<code>GrB_COMP</code>	Use the complement of <b>mask</b> .

## Return Values

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

**GrB\_PANIC** Unknown internal error.

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

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

3764 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
 3765 a call to `new` (or `dup` for vector parameters).

3766 GrB\_DIMENSION\_MISMATCH Mask or vector dimensions are incompatible.

3767 GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with the cor-  
 3768 responding domains of the binary operator (`op`) or accumulation  
 3769 operator, or the mask's domain is not compatible with `bool` (in the  
 3770 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3771 Description

3772 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS vectors:  $w =$   
 3773  $u \oplus v$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $w = w \odot (u \oplus v)$ . Logically,  
 3774 this operation occurs in three steps:

3775 **Setup** The internal vectors and mask used in the computation are formed and their domains  
 3776 and dimensions are tested for compatibility.

3777 **Compute** The indicated computations are carried out.

3778 **Output** The result is written into the output vector, possibly under control of a mask.

3779 Up to four argument vectors are used in the `GrB_eWiseAdd` operation:

- 3780 1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3781 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3782 3.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$
- 3783 4.  $v = \langle \mathbf{D}(v), \mathbf{size}(v), \mathbf{L}(v) = \{(i, v_i)\} \rangle$

3784 The argument vectors, the “sum” operator (`op`), and the accumulation operator (if provided) are  
 3785 tested for domain compatibility as follows:

- 3786 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
 3787 must be from one of the pre-defined types of Table 3.2.
- 3788 2.  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$ .
- 3789 3.  $\mathbf{D}(v)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$ .
- 3790 4.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3791 5.  $\mathbf{D}(u)$  and  $\mathbf{D}(v)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3792 6. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 3793 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
 3794 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 vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$  denotes copy):

1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - (b) If `mask  $\neq$  GrB_NULL`,
    - i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
  - (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
4. Vector  $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$ .

The internal vectors and mask are checked for dimension compatibility. The following conditions must hold:

1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}}) = \mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{v}})$ .

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 vectors:

- $\tilde{\mathbf{t}}$ : The vector holding the element-wise “sum” of  $\tilde{\mathbf{u}}$  and vector  $\tilde{\mathbf{v}}$ .
- $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector  $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, t_i) : \mathbf{ind}(\tilde{\mathbf{u}}) \cup \mathbf{ind}(\tilde{\mathbf{v}}) \neq \emptyset\} \rangle$  is created. The value of each of its elements is computed by:

$$t_i = (\tilde{\mathbf{u}}(i) \oplus \tilde{\mathbf{v}}(i)), \forall i \in (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}}))$$

$$t_i = \tilde{\mathbf{u}}(i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{u}}) - (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}})))$$

3827  
3828

$$t_i = \tilde{\mathbf{v}}(i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{v}}) - (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}})))$$

3829 where the difference operator in the previous expressions refers to set difference.

3830 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3831 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 3832 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3834 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
3835 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 3836 z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 3837 \\ 3838 z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3839 \\ 3840 z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

3841 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3842 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
3843 using what is called a *standard vector mask and replace*. This is carried out under control of the  
3844 mask which acts as a “write mask”.

- 3845 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
3846 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3848 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
3849 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
3850 mask are unchanged:

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3852 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
3853 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
3854 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but  
3855 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
3856 sequence.

#### 3857 4.3.5.2 eWiseAdd: Matrix variant

3858 Perform element-wise (general) addition on the elements of two matrices, producing a third matrix  
3859 as result.

## 3860 C Syntax

```

3861     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3862                           const GrB_Matrix Mask,
3863                           const GrB_BinaryOp accum,
3864                           const GrB_Semiring op,
3865                           const GrB_Matrix A,
3866                           const GrB_Matrix B,
3867                           const GrB_Descriptor desc);
3868
3869     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3870                           const GrB_Matrix Mask,
3871                           const GrB_BinaryOp accum,
3872                           const GrB_Monoid op,
3873                           const GrB_Matrix A,
3874                           const GrB_Matrix B,
3875                           const GrB_Descriptor desc);
3876
3877     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3878                           const GrB_Matrix Mask,
3879                           const GrB_BinaryOp accum,
3880                           const GrB_BinaryOp op,
3881                           const GrB_Matrix A,
3882                           const GrB_Matrix B,
3883                           const GrB_Descriptor desc);

```

## 3884 Parameters

3885 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
3886 that may be accumulated with the result of the element-wise operation. On output,  
3887 the matrix holds the results of the operation.

3888 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
3889 stored into the output matrix C. The mask dimensions must match those of the  
3890 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
3891 of the **Mask** matrix must be of type `bool` or any of the predefined “built-in” types  
3892 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
3893 dimensions of C), `GrB_NULL` should be specified.

3894 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
3895 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
3896 specified.

3897 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”  
3898 operation. Depending on which type is passed, the following defines the binary  
3899 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$ , used:

3900 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .  
 3901 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-  
 3902 nored.  
 3903 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-  
 3904 nary op and additive identity are ignored.

3905 A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
 3906 operation.

3907 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
 3908 operation.

3909 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 3910 should be specified. Non-default field/value pairs are listed as follows:  
 3911

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.

## 3913 Return Values

3914 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 3915 blocking mode, this indicates that the compatibility tests on di-  
 3916 mensions and domains for the input arguments passed successfully.  
 3917 Either way, output matrix C is ready to be used in the next method  
 3918 of the sequence.

3919 GrB\_PANIC Unknown internal error.

3920 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 3921 GraphBLAS objects (input or output) is in an invalid state caused  
 3922 by a previous execution error. Call GrB\_error() to access any error  
 3923 messages generated by the implementation.

3924 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

3925 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
 3926 a call to new (or Matrix\_dup for matrix parameters).

3927 GrB\_DIMENSION\_MISMATCH Mask and/or matrix dimensions are incompatible.



3928 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
 3929 corresponding domains of the binary operator ( $\text{op}$ ) or accumulation  
 3930 operator, or the mask's domain is not compatible with `bool` (in the  
 3931 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3932 Description

3933 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS matrices:  
 3934  $C = A \oplus B$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot (A \oplus B)$ .  
 3935 Logically, this operation occurs in three steps:

3936 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 3937 and dimensions are tested for compatibility.

3938 **Compute** The indicated computations are carried out.

3939 **Output** The result is written into the output matrix, possibly under control of a mask.

3940 Up to four argument matrices are used in the `GrB_eWiseAdd` operation:

- 3941 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3942 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)
- 3943 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3944 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3945 The argument matrices, the “sum” operator ( $\text{op}$ ), and the accumulation operator (if provided) are  
 3946 tested for domain compatibility as follows:

- 3947 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 3948 must be from one of the pre-defined types of Table 3.2.
- 3949 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$ .
- 3950 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$ .
- 3951 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3952 5.  $\mathbf{D}(A)$  and  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3953 6. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 3954 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of  $\text{op}$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
 3955 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  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
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.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$ .
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.

3987 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cup \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\}$   
 3988 is created. The value of each of its elements is computed by

$$\begin{aligned} 3989 \quad T_{ij} &= (\tilde{\mathbf{A}}(i, j) \oplus \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \\ 3990 \quad T_{ij} &= \tilde{\mathbf{A}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{A}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}))) \\ 3991 \quad T_{ij} &= \tilde{\mathbf{B}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{B}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}))) \end{aligned}$$

3994 where the difference operator in the previous expressions refers to set difference.

3995 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 3996 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 3997 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$3998 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3999 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 4000 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 4001 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 4002 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4003 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4004 \quad & \\ 4005 \end{aligned}$$

4006 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4007 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 4008 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 4009 mask which acts as a “write mask”.

- 4010 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
 4011 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$4012 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 4013 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 4014 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 4015 mask are unchanged:

$$4016 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4017 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
 4018 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
 4019 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
 4020 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 4021 sequence.

### 4022 4.3.6 extract: Selecting sub-graphs

4023 Extract a subset of a matrix or vector.

#### 4024 4.3.6.1 extract: Standard vector variant

4025 Extract a sub-vector from a larger vector as specified by a set of indices. The result is a vector  
4026 whose size is equal to the number of indices.

### 4027 C Syntax

```
4028         GrB_Info GrB_extract(GrB_Vector          w,  
4029                             const GrB_Vector      mask,  
4030                             const GrB_BinaryOp     accum,  
4031                             const GrB_Vector      u,  
4032                             const GrB_Index       *indices,  
4033                             GrB_Index             nindices,  
4034                             const GrB_Descriptor   desc);
```

### 4035 Parameters

4036 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
4037 that may be accumulated with the result of the extract operation. On output, this  
4038 vector holds the results of the operation.

4039 **mask** (IN) An optional “write” mask that controls which results from this operation are  
4040 stored into the output vector **w**. The mask dimensions must match those of the  
4041 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4042 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
4043 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4044 dimensions of **w**), **GrB\_NULL** should be specified.

4045 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
4046 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4047 specified.

4048 **u** (IN) The GraphBLAS vector from which the subset is extracted.

4049 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations of  
4050 elements from **u** that are extracted. If all elements of **u** are to be extracted in order  
4051 from 0 to **nindices** – 1, then **GrB\_ALL** should be specified. Regardless of execution  
4052 mode and return value, this array may be manipulated by the caller after this  
4053 operation returns without affecting any deferred computations for this operation.

4054 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(w)**.

4055 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 4056 should be specified. Non-default field/value pairs are listed as follows:

4057

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector <b>w</b> is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of <b>mask</b> .

4058

## 4059 Return Values

4060 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 4061 blocking mode, this indicates that the compatibility tests on  
 4062 dimensions and domains for the input arguments passed suc-  
 4063 cessfully. Either way, output vector **w** is ready to be used in the  
 4064 next method of the sequence.

4065 GrB\_PANIC Unknown internal error.

4066 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
 4067 opaque GraphBLAS objects (input or output) is in an invalid  
 4068 state caused by a previous execution error. Call GrB\_error() to  
 4069 access any error messages generated by the implementation.

4070 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

4071 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
 4072 by a call to **new** (or **dup** for vector parameters).

4073 GrB\_INDEX\_OUT\_OF\_BOUNDS A value in **indices** is greater than or equal to **size(u)**. In non-  
 4074 blocking mode, this error can be deferred.

4075 GrB\_DIMENSION\_MISMATCH **mask** and **w** dimensions are incompatible, or **nindices**  $\neq$  **size(w)**.

4076 GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with each  
 4077 other or the corresponding domains of the accumulation oper-  
 4078 ator, or the mask's domain is not compatible with **bool** (in the  
 4079 case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

4080 GrB\_NULL\_POINTER Argument **row\_indices** is a NULL pointer.

## 4081 Description

4082 This variant of GrB\_extract computes the result of extracting a subset of locations from a Graph-  
 4083 BLAS vector in a specific order: **w** = **u(indices)**; or, if an optional binary accumulation operator

4084  $(\odot)$  is provided,  $w = w \odot u(\text{indices})$ . More explicitly:

$$4085 \quad \begin{aligned} w(i) &= u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ w(i) &= w(i) \odot u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices} \end{aligned}$$

4086 Logically, this operation occurs in three steps:

4087     **Setup** The internal vectors and mask used in the computation are formed and their domains  
4088             and dimensions are tested for compatibility.

4089     **Compute** The indicated computations are carried out.

4090     **Output** The result is written into the output vector, possibly under control of a mask.

4091 Up to three argument vectors are used in this GrB\_extract operation:

- 4092     1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4093     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4094     3.  $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

4095 The argument vectors and the accumulation operator (if provided) are tested for domain compati-  
4096 bility as follows:

- 4097     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
4098         must be from one of the pre-defined types of Table 3.2.
- 4099     2.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}(u)$ .
- 4100     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
4101         of the accumulation operator and  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
4102         mulation operator.

4103 Two domains are compatible with each other if values from one domain can be cast to values in  
4104 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4105 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4106 any compatibility rule above is violated, execution of GrB\_extract ends and the domain mismatch  
4107 error listed above is returned.

4108 From the arguments, the internal vectors, mask, and index array used in the computation are  
4109 formed ( $\leftarrow$  denotes copy):

- 4110     1. Vector  $\tilde{w} \leftarrow w$ .
- 4111     2. One-dimensional mask,  $\tilde{m}$ , is computed from argument `mask` as follows:  
4112         (a) If `mask = GrB_NULL`, then  $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$ .

- 4113 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,
- 4114 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
- 4115 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
- 4116 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .
- 4117 3. Vector  $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 4118 4. The internal index array,  $\widetilde{\mathbf{I}}$ , is computed from argument indices as follows:
- 4119 (a) If  $\text{indices} = \text{GrB\_ALL}$ , then  $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$ .
- 4120 (b) Otherwise,  $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$ .

4121 The internal vectors and mask are checked for dimension compatibility. The following conditions  
4122 must hold:

- 4123 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 4124 2.  $\text{nindices} = \text{size}(\widetilde{\mathbf{w}})$ .

4125 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
4126 match error listed above is returned.

4127 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4128 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4129 We are now ready to carry out the extract and any additional associated operations. We describe  
4130 this in terms of two intermediate vectors:

- 4131 •  $\widetilde{\mathbf{t}}$ : The vector holding the extraction from  $\widetilde{\mathbf{u}}$  in their destination locations relative to  $\widetilde{\mathbf{w}}$ .
- 4132 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4133 The intermediate vector,  $\widetilde{\mathbf{t}}$ , is created as follows:

$$4134 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{w}}), \{(i, \widetilde{\mathbf{u}}[\widetilde{\mathbf{I}}[i]]) \mid \forall i, 0 \leq i < \text{nindices} : \widetilde{\mathbf{I}}[i] \in \text{ind}(\widetilde{\mathbf{u}})\} \rangle.$$

4135 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  
4136 `GrB_extract` ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING`  
4137 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
4138 result vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4139 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 4140 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$ .
- 4141 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{z}}$  is defined as

$$4142 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ , using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a “write mask”.

- If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

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

#### 4.3.6.2 extract: Standard matrix variant

Extract a sub-matrix from a larger matrix as specified by a set of row indices and a set of column indices. The result is a matrix whose size is equal to size of the sets of indices.

### C Syntax

```
GrB_Info GrB_extract(GrB_Matrix      C,
                    const GrB_Matrix  Mask,
                    const GrB_BinaryOp accum,
                    const GrB_Matrix  A,
                    const GrB_Index   *row_indices,
                    GrB_Index         nrows,
                    const GrB_Index   *col_indices,
                    GrB_Index         ncols,
                    const GrB_Descriptor desc);
```



## Parameters

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

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

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

**A** (IN) The GraphBLAS matrix from which the subset is extracted.

**row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **A** from which elements are extracted. If elements in all rows of **A** are to be extracted in order, **GrB\_ALL** should be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation.

**nrows** (IN) The number of values in the **row\_indices** array. Must be equal to **nrows(C)**.

**col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns of **A** from which elements are extracted. If elements in all columns of **A** are to be extracted in order, then **GrB\_ALL** should be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation.

**ncols** (IN) The number of values in the **col\_indices** array. Must be equal to **ncols(C)**.

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

Param	Field	Value	Description
<b>C</b>	<b>GrB_OUTP</b>	<b>GrB_REPLACE</b>	Output matrix <b>C</b> is cleared (all elements removed) before the result is stored in it.
<b>Mask</b>	<b>GrB_MASK</b>	<b>GrB_STRUCTURE</b>	The write mask is constructed from the structure (pattern of stored values) of the input <b>Mask</b> matrix. The stored values are not examined.
<b>Mask</b>	<b>GrB_MASK</b>	<b>GrB_COMP</b>	Use the complement of <b>Mask</b> .
<b>A</b>	<b>GrB_INP0</b>	<b>GrB_TRAN</b>	Use transpose of <b>A</b> for the operation.

## 4209 Return Values

4210	<b>GrB_SUCCESS</b>	In blocking mode, the operation completed successfully. In non-
4211		blocking mode, this indicates that the compatibility tests on
4212		dimensions and domains for the input arguments passed suc-
4213		cessfully. Either way, output matrix C is ready to be used in the
4214		next method of the sequence.
4215	<b>GrB_PANIC</b>	Unknown internal error.
4216	<b>GrB_INVALID_OBJECT</b>	This is returned in any execution mode whenever one of the
4217		opaque GraphBLAS objects (input or output) is in an invalid
4218		state caused by a previous execution error. Call <code>GrB_error()</code> to
4219		access any error messages generated by the implementation.
4220	<b>GrB_OUT_OF_MEMORY</b>	Not enough memory available for the operation.
4221	<b>GrB_UNINITIALIZED_OBJECT</b>	One or more of the GraphBLAS objects has not been initialized
4222		by a call to <code>new</code> (or <code>Matrix_dup</code> for matrix parameters).
4223	<b>GrB_INDEX_OUT_OF_BOUNDS</b>	A value in <code>row_indices</code> is greater than or equal to <code>nrows(A)</code> , or
4224		a value in <code>col_indices</code> is greater than or equal to <code>ncols(A)</code> . In
4225		non-blocking mode, this error can be deferred.
4226	<b>GrB_DIMENSION_MISMATCH</b>	Mask and C dimensions are incompatible, <code>nrows</code> $\neq$ <code>nrows(C)</code> , or
4227		<code>ncols</code> $\neq$ <code>ncols(C)</code> .
4228	<b>GrB_DOMAIN_MISMATCH</b>	The domains of the various matrices are incompatible with each
4229		other or the corresponding domains of the accumulation oper-
4230		ator, or the mask's domain is not compatible with <code>bool</code> (in the
4231		case where <code>desc[GrB_MASK].GrB_STRUCTURE</code> is not set).
4232	<b>GrB_NULL_POINTER</b>	Either argument <code>row_indices</code> is a NULL pointer, argument <code>col_indices</code>
4233		is a NULL pointer, or both.

## 4234 Description

4235 This variant of `GrB_extract` computes the result of extracting a subset of locations from specified  
 4236 rows and columns of a GraphBLAS matrix in a specific order:  $C = A(\text{row\_indices}, \text{col\_indices})$ ; or,  
 4237 if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot A(\text{row\_indices}, \text{col\_indices})$ .  
 4238 More explicitly (not accounting for an optional transpose of A):

$$\begin{aligned}
 &C(i, j) = A(\text{row\_indices}[i], \text{col\_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\
 &C(i, j) = C(i, j) \odot A(\text{row\_indices}[i], \text{col\_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}
 \end{aligned}$$

4240 Logically, this operation occurs in three steps:

4241 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 4242 and dimensions are tested for compatibility.

4243 **Compute** The indicated computations are carried out.

4244 **Output** The result is written into the output matrix, possibly under control of a mask.

4245 Up to three argument matrices are used in the `GrB_extract` operation:

- 4246 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4247 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)
- 4248 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4249 The argument matrices and the accumulation operator (if provided) are tested for domain compat-  
4250 ibility as follows:

- 4251 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
4252 must be from one of the pre-defined types of Table 3.2.
- 4253 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$ .
- 4254 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
4255 of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
4256 mulation operator.

4257 Two domains are compatible with each other if values from one domain can be cast to values in  
4258 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4259 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4260 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch  
4261 error listed above is returned.

4262 From the arguments, the internal matrices, `mask`, and index arrays used in the computation are  
4263 formed ( $\leftarrow$  denotes copy):

- 4264 1. Matrix  $\tilde{C} \leftarrow C$ .
- 4265 2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument `Mask` as follows:
  - 4266 (a) If `Mask` = `GrB_NULL`, then  $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$   
4267  $j < \mathbf{ncols}(C)\} \rangle$ .
  - 4268 (b) If `Mask`  $\neq$  `GrB_NULL`,
    - 4269 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
4270  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
    - 4271 ii. Otherwise,  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
4272  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
  - 4273 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{M} \leftarrow \neg \tilde{M}$ .
- 4274 3. Matrix  $\tilde{A} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? A^T : A$ .

- 4275 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
- 4276 (a) If `row_indices` = `GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$ .
- 4277 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{row\_indices}[i], \forall i : 0 \leq i < \text{nrows}$ .
- 4278 5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:
- 4279 (a) If `col_indices` = `GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \text{ncols}$ .
- 4280 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \text{col\_indices}[j], \forall j : 0 \leq j < \text{ncols}$ .

4281 The internal matrices and mask are checked for dimension compatibility. The following conditions  
4282 must hold:

- 4283 1.  $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}(\tilde{\mathbf{M}})$ .
- 4284 2.  $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}(\tilde{\mathbf{M}})$ .
- 4285 3.  $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}$ .
- 4286 4.  $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}$ .

4287 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
4288 match error listed above is returned.

4289 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4290 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4291 We are now ready to carry out the extract and any additional associated operations. We describe  
4292 this in terms of two intermediate matrices:

- 4293 •  $\tilde{\mathbf{T}}$ : The matrix holding the extraction from  $\tilde{\mathbf{A}}$ .
- 4294 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

4295 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

4296 
$$\tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}),$$

$$\{(i, j, \tilde{\mathbf{A}}(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j])) \mid \forall (i, j), 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} : (\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j]) \in \text{ind}(\tilde{\mathbf{A}})\} \rangle.$$

4297 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  $\tilde{\mathbf{J}}$   
4298 array is not in the range  $[0, \text{ncols}(\tilde{\mathbf{A}}))$ , the execution of `GrB_extract` ends and the index out-of-  
4299 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred  
4300 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from  
4301 this point forward in the sequence.

4302 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 4303 • If `accum` = `GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .

- If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$\tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \text{ind}(\tilde{\mathbf{C}}) \cup \text{ind}(\tilde{\mathbf{T}})\} \rangle.$$

The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\text{ind}(\tilde{\mathbf{T}}) \cap \text{ind}(\tilde{\mathbf{C}})),$$

$$Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\text{ind}(\tilde{\mathbf{C}}) - (\text{ind}(\tilde{\mathbf{T}}) \cap \text{ind}(\tilde{\mathbf{C}}))),$$

$$Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\text{ind}(\tilde{\mathbf{T}}) - (\text{ind}(\tilde{\mathbf{T}}) \cap \text{ind}(\tilde{\mathbf{C}}))),$$

where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ , using what is called a *standard matrix mask and replace*. This is carried out under control of the mask which acts as a “write mask”.

- If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$\mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\text{ind}(\tilde{\mathbf{Z}}) \cap \text{ind}(\tilde{\mathbf{M}}))\}.$$

- If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\text{ind}(\mathbf{C}) \cap \text{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\text{ind}(\tilde{\mathbf{Z}}) \cap \text{ind}(\tilde{\mathbf{M}}))\}.$$

In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

#### 4.3.6.3 extract: Column (and row) variant

Extract from one column of a matrix into a vector. Note that with the transpose descriptor for the source matrix, elements of an arbitrary row of the matrix can be extracted with this function as well.

## 4333 C Syntax

```

4334      GrB_Info GrB_extract(GrB_Vector      w,
4335                          const GrB_Vector mask,
4336                          const GrB_BinaryOp accum,
4337                          const GrB_Matrix A,
4338                          const GrB_Index *row_indices,
4339                          GrB_Index nrows,
4340                          GrB_Index col_index,
4341                          const GrB_Descriptor desc);

```

## 4342 Parameters

4343     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
4344     that may be accumulated with the result of the extract operation. On output, this  
4345     vector holds the results of the operation.

4346     **mask** (IN) An optional “write” mask that controls which results from this operation are  
4347     stored into the output vector **w**. The mask dimensions must match those of the  
4348     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4349     of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
4350     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4351     dimensions of **w**), **GrB\_NULL** should be specified.

4352     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
4353     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4354     specified.

4355     **A** (IN) The GraphBLAS matrix from which the column subset is extracted.

4356     **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations  
4357     within the specified column of **A** from which elements are extracted. If elements in  
4358     all rows of **A** are to be extracted in order, **GrB\_ALL** should be specified. Regardless  
4359     of execution mode and return value, this array may be manipulated by the caller  
4360     after this operation returns without affecting any deferred computations for this  
4361     operation.

4362     **nrows** (IN) The number of indices in the **row\_indices** array. Must be equal to **size(w)**.

4363     **col\_index** (IN) The index of the column of **A** from which to extract values. It must be in the  
4364     range  $[0, \mathbf{ncols}(A))$ .

4365     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
4366     should be specified. Non-default field/value pairs are listed as follows:

4367

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector <b>w</b> is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of <b>mask</b> .
A	GrB_INP0	GrB_TRAN	Use transpose of <b>A</b> for the operation.

## Return Values

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

**GrB\_PANIC** Unknown internal error.

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

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

**GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by a call to **new** (or **dup** for vector or matrix parameters).

**GrB\_INVALID\_INDEX** **col\_index** is outside the allowable range (i.e., greater than **ncols(A)**).

**GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **row\_indices** is greater than or equal to **nrows(A)**. In non-blocking mode, this error can be deferred.

**GrB\_DIMENSION\_MISMATCH** **mask** and **w** dimensions are incompatible, or **nrows**  $\neq$  **size(w)**.

**GrB\_DOMAIN\_MISMATCH** The domains of the vector or matrix are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with **bool** (in the case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

**GrB\_NULL\_POINTER** Argument **row\_indices** is a NULL pointer.

## Description

This variant of **GrB\_extract** computes the result of extracting a subset of locations (in a specific order) from a specified column of a GraphBLAS matrix: **w** = **A(:, col\_index)(row\_indices)**; or, if

4395 an optional binary accumulation operator ( $\odot$ ) is provided,  $w = w \odot A(:, \text{col\_index})(\text{row\_indices})$ .  
 4396 More explicitly:

$$4397 \quad \begin{aligned} w(i) &= A(\text{row\_indices}[i], \text{col\_index}) \quad \forall i : 0 \leq i < \text{nrows}, \quad \text{or} \\ w(i) &= w(i) \odot A(\text{row\_indices}[i], \text{col\_index}) \quad \forall i : 0 \leq i < \text{nrows} \end{aligned}$$

4398 Logically, this operation occurs in three steps:

4399     **Setup** The internal matrices, vectors, and mask used in the computation are formed and their  
 4400 domains and dimensions are tested for compatibility.

4401     **Compute** The indicated computations are carried out.

4402     **Output** The result is written into the output vector, possibly under control of a mask.

4403 Up to three argument vectors and matrices are used in this GrB\_extract operation:

- 4404     1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4405     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4406     3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4407 The argument vectors, matrix and the accumulation operator (if provided) are tested for domain  
 4408 compatibility as follows:

- 4409     1. If **mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{mask})$   
 4410 must be from one of the pre-defined types of Table 3.2.
- 4411     2.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}(A)$ .
- 4412     3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 4413 of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 4414 mulation operator.

4415 Two domains are compatible with each other if values from one domain can be cast to values in  
 4416 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 4417 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 4418 any compatibility rule above is violated, execution of GrB\_extract ends and the domain mismatch  
 4419 error listed above is returned.

4420 From the arguments, the internal vector, matrix, mask, and index array used in the computation  
 4421 are formed ( $\leftarrow$  denotes copy):

- 4422     1. Vector  $\tilde{w} \leftarrow w$ .
- 4423     2. One-dimensional mask,  $\tilde{m}$ , is computed from argument **mask** as follows:  
 4424         (a) If **mask** = GrB\_NULL, then  $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$ .



4425 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,  
 4426 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,  
 4427 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .  
 4428 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .  
 4429 3. Matrix  $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .  
 4430 4. The internal row index array,  $\widetilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:  
 4431 (a) If `indices = GrB_ALL`, then  $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$ .  
 4432 (b) Otherwise,  $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nrows}$ .

4433 The internal vector, `mask`, and index array are checked for dimension compatibility. The following  
 4434 conditions must hold:

- 4435 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 4436 2.  $\text{size}(\widetilde{\mathbf{w}}) = \text{nrows}$ .

4437 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
 4438 match error listed above is returned.

4439 The `col_index` parameter is checked for a valid value. The following condition must hold:

- 4440 1.  $0 \leq \text{col\_index} < \text{ncols}(\mathbf{A})$

4441 If the rule above is violated, execution of `GrB_extract` ends and the invalid index error listed above  
 4442 is returned.

4443 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 4444 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4445 We are now ready to carry out the extract and any additional associated operations. We describe  
 4446 this in terms of two intermediate vectors:

- 4447 •  $\widetilde{\mathbf{t}}$ : The vector holding the extraction from a column of  $\widetilde{\mathbf{A}}$ .
- 4448 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4449 The intermediate vector,  $\widetilde{\mathbf{t}}$ , is created as follows:

$$4450 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}, \{(i, \widetilde{\mathbf{A}}(\widetilde{\mathbf{I}}[i], \text{col\_index})) \mid \forall i, 0 \leq i < \text{nrows} : (\widetilde{\mathbf{I}}[i], \text{col\_index}) \in \text{ind}(\widetilde{\mathbf{A}})\} \rangle.$$

4451 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`  
 4452 ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING` mode,  
 4453 the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the result  
 4454 vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4455 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

4456 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .

4457 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$4458 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4459 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
4460 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$4461 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

$$4462 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

$$4463 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

4464 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4467 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
4468 using what is called a *standard vector mask and replace*. This is carried out under control of the  
4469 mask which acts as a “write mask”.

4470 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
4471 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$4472 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4473 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
4474 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
4475 mask are unchanged:

$$4476 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4477 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
4478 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
4479 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but  
4480 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
4481 sequence.

### 4482 4.3.7 assign: Modifying sub-graphs

4483 Assign the contents of a subset of a matrix or vector.

#### 4484 4.3.7.1 assign: Standard vector variant

4485 Assign values from one GraphBLAS vector to a subset of a vector as specified by a set of indices.  
4486 The size of the input vector is the same size as the index array provided.

## 4487 C Syntax

```
4488         GrB_Info GrB_assign(GrB_Vector      w,  
4489                             const GrB_Vector mask,  
4490                             const GrB_BinaryOp accum,  
4491                             const GrB_Vector u,  
4492                             const GrB_Index *indices,  
4493                             GrB_Index nindices,  
4494                             const GrB_Descriptor desc);
```

## 4495 Parameters

4496 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
4497 that may be accumulated with the result of the assign operation. On output, this  
4498 vector holds the results of the operation.

4499 **mask** (IN) An optional “write” mask that controls which results from this operation are  
4500 stored into the output vector **w**. The mask dimensions must match those of the  
4501 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4502 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
4503 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4504 dimensions of **w**), **GrB\_NULL** should be specified.

4505 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
4506 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4507 specified.

4508 **u** (IN) The GraphBLAS vector whose contents are assigned to a subset of **w**.

4509 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
4510 **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0  
4511 to **nindices** – 1, then **GrB\_ALL** should be specified. Regardless of execution mode  
4512 and return value, this array may be manipulated by the caller after this operation  
4513 returns without affecting any deferred computations for this operation. If this  
4514 array contains duplicate values, it implies in assignment of more than one value to  
4515 the same location which leads to undefined results.

4516 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(u)**.

4517 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
4518 should be specified. Non-default field/value pairs are listed as follows:  
4519

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:

$$\begin{aligned}
 w(\text{indices}[i]) &= u(i), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\
 w(\text{indices}[i]) &= w(\text{indices}[i]) \odot u(i), \forall i : 0 \leq i < \text{nindices}.
 \end{aligned}$$

4548 Logically, this operation occurs in three steps:

4549     **Setup** The internal vectors and mask used in the computation are formed and their domains  
4550             and dimensions are tested for compatibility.

4551     **Compute** The indicated computations are carried out.

4552     **Output** The result is written into the output vector, possibly under control of a mask.

4553 Up to three argument vectors are used in the `GrB_assign` operation:

- 4554     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 4555     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4556     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4557 The argument vectors and the accumulation operator (if provided) are tested for domain compati-  
4558 bility as follows:

- 4559     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
4560         must be from one of the pre-defined types of Table 3.2.
- 4561     2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .
- 4562     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
4563         of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accu-  
4564         mulation operator.

4565 Two domains are compatible with each other if values from one domain can be cast to values in  
4566 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4567 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4568 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
4569 error listed above is returned.

4570 From the arguments, the internal vectors, mask and index array used in the computation are formed  
4571 ( $\leftarrow$  denotes copy):

- 4572     1. Vector  $\widetilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 4573     2. One-dimensional mask,  $\widetilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 4574         (a) If `mask` = `GrB_NULL`, then  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 4575         (b) If `mask`  $\neq$  `GrB_NULL`,
    - 4576             i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 4577             ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
  - 4578         (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .

4579 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

4580 4. The internal index array,  $\tilde{\mathbf{I}}$ , is computed from argument indices as follows:

4581 (a) If `indices = GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$ .

4582 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$ .

4583 The internal vector and mask are checked for dimension compatibility. The following conditions  
4584 must hold:

4585 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

4586 2.  $\text{nindices} = \text{size}(\tilde{\mathbf{u}})$ .

4587 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
4588 match error listed above is returned.

4589 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4590 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4591 We are now ready to carry out the assign and any additional associated operations. We describe  
4592 this in terms of two intermediate vectors:

- 4593 •  $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{w}}$ .
- 4594 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4595 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$4596 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nindices} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4597 At this point, if any value of  $\tilde{\mathbf{I}}[i]$  is outside the valid range of indices for vector  $\tilde{\mathbf{w}}$ , computation  
4598 ends and the method returns the index-out-of-bounds error listed above. In `GrB_NONBLOCKING`  
4599 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
4600 result vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4601 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

- 4602 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$4603 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4604 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
4605 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  
4606 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\text{ind}(\tilde{\mathbf{t}})$ ).

4607 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
4608 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$4609 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))),$$

$$4610 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4612 where the difference operator refers to set difference.

4613 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$4614 \quad \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4615 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
4616 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 4617 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 4618 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 4619 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

4622 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4623 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
4624 using what is called a *standard vector mask and replace*. This is carried out under control of the  
4625 mask which acts as a “write mask”.

4626 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
4627 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$4628 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4629 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
4630 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
4631 mask are unchanged:

$$4632 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4633 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
4634 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
4635 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but  
4636 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
4637 sequence.

#### 4638 4.3.7.2 assign: Standard matrix variant

4639 Assign values from one GraphBLAS matrix to a subset of a matrix as specified by a set of indices.  
4640 The dimensions of the input matrix are the same size as the row and column index arrays provided.

### 4641 C Syntax

```
4642      GrB_Info GrB_assign(GrB_Matrix      C,
4643                          const GrB_Matrix Mask,
4644                          const GrB_BinaryOp accum,
4645                          const GrB_Matrix A,
```

```

4646         const GrB_Index      *row_indices,
4647         GrB_Index             nrows,
4648         const GrB_Index      *col_indices,
4649         GrB_Index             ncols,
4650         const GrB_Descriptor  desc);

```

## 4651 Parameters

4652     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
4653     that may be accumulated with the result of the assign operation. On output, the  
4654     matrix holds the results of the operation.

4655     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
4656     stored into the output matrix **C**. The mask dimensions must match those of the  
4657     matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4658     of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
4659     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4660     dimensions of **C**), **GrB\_NULL** should be specified.

4661     **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
4662     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4663     specified.

4664     **A** (IN) The GraphBLAS matrix whose contents are assigned to a subset of **C**.

4665     **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**  
4666     that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** – 1,  
4667     then **GrB\_ALL** can be specified. Regardless of execution mode and return value,  
4668     this array may be manipulated by the caller after this operation returns without  
4669     affecting any deferred computations for this operation. If this array contains du-  
4670     plicate values, it implies assignment of more than one value to the same location  
4671     which leads to undefined results.

4672     **nrows** (IN) The number of values in the **row\_indices** array. Must be equal to **nrows(A)**  
4673     if **A** is not transposed, or equal to **ncols(A)** if **A** is transposed.

4674     **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns  
4675     of **C** that are assigned. If all columns of **C** are to be assigned in order from 0  
4676     to **ncols** – 1, then **GrB\_ALL** should be specified. Regardless of execution mode  
4677     and return value, this array may be manipulated by the caller after this operation  
4678     returns without affecting any deferred computations for this operation. If this  
4679     array contains duplicate values, it implies assignment of more than one value to  
4680     the same location which leads to undefined results.

4681     **ncols** (IN) The number of values in **col\_indices** array. Must be equal to **ncols(A)** if **A** is  
4682     not transposed, or equal to **nrows(A)** if **A** is transposed.



4683  
4684  
4685

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:

4686

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.

## 4687 Return Values

4688  
4689  
4690  
4691  
4692

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.

4693

GrB\_PANIC Unknown internal error.

4694  
4695  
4696  
4697

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.

4698

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

4699  
4700

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).

4701  
4702  
4703

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.

4704  
4705

GrB\_DIMENSION\_MISMATCH Mask and C dimensions are incompatible, nrow  $\neq$  nrow(A), or ncols  $\neq$  ncols(A).

4706  
4707  
4708  
4709

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).

4710  
4711

GrB\_NULL\_POINTER Either argument row\_indices is a NULL pointer, argument col\_indices is a NULL pointer, or both.

## 4712 Description

4713 This variant of `GrB_assign` computes the result of assigning the contents of `A` to a subset of rows  
 4714 and columns in `C` in a specified order:  $C(\text{row\_indices}, \text{col\_indices}) = A$ ; or, if an optional binary  
 4715 accumulation operator ( $\odot$ ) is provided,  $C(\text{row\_indices}, \text{col\_indices}) = C(\text{row\_indices}, \text{col\_indices}) \odot$   
 4716 `A`. More explicitly (not accounting for an optional transpose of `A`):

$$\begin{aligned} & C(\text{row\_indices}[i], \text{col\_indices}[j]) = A(i, j), \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\ 4717 & C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot A(i, j), \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

4718 Logically, this operation occurs in three steps:

4719     Setup The internal matrices and mask used in the computation are formed and their domains  
 4720     and dimensions are tested for compatibility.

4721     Compute The indicated computations are carried out.

4722     Output The result is written into the output matrix, possibly under control of a mask.

4723 Up to three argument matrices are used in the `GrB_assign` operation:

- 4724 1.  $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4725 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)
- 4726 3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4727 The argument matrices and the accumulation operator (if provided) are tested for domain compat-  
 4728 ibility as follows:

- 4729 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 4730 must be from one of the pre-defined types of Table 3.2.
- 4731 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$ .
- 4732 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 4733 of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 4734 mulation operator.

4735 Two domains are compatible with each other if values from one domain can be cast to values in  
 4736 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 4737 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 4738 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
 4739 error listed above is returned.

4740 From the arguments, the internal matrices, mask, and index arrays used in the computation are  
 4741 formed ( $\leftarrow$  denotes copy):

- 4742 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 4743 2. Two-dimensional mask  $\tilde{\mathbf{M}}$  is computed from argument `Mask` as follows:
- 4744 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
4745  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
- 4746 (b) If `Mask  $\neq$  GrB_NULL`,
- 4747 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
4748  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
- 4749 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
4750  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
- 4751 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 4752 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 4753 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
- 4754 (a) If `row_indices = GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .
- 4755 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{row\_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .
- 4756 5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:
- 4757 (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$ .
- 4758 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \mathbf{col\_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$ .

4759 The internal matrices and mask are checked for dimension compatibility. The following conditions  
4760 must hold:

- 4761 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 4762 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 4763 3.  $\mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}$ .
- 4764 4.  $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}$ .

4765 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
4766 match error listed above is returned.

4767 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4768 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4769 We are now ready to carry out the assign and any additional associated operations. We describe  
4770 this in terms of two intermediate vectors:

- 4771 •  $\tilde{\mathbf{T}}$ : The matrix holding the contents from  $\tilde{\mathbf{A}}$  in their destination locations relative to  $\tilde{\mathbf{C}}$ .
- 4772 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

4773 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$4774 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \tilde{\mathbf{A}}(i, j)) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols} : (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle.$$

4775 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  
4776  $\tilde{\mathbf{J}}$  array is not in the range  $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$ , the execution of `GrB_assign` ends and the index out-of-  
4777 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred  
4778 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from  
4779 this point forward in the sequence.

4780 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows:

- 4781 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}}$  is defined as

$$4782 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 4783 \quad \{(i, j, Z_{ij}) \mid \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4784 The above expression defines the structure of matrix  $\tilde{\mathbf{Z}}$  as follows: We start with the structure  
4785 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  
4786 assigned ( $\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$ ). Finally, we add the structure of  $\tilde{\mathbf{T}}$  ( $\mathbf{ind}(\tilde{\mathbf{T}})$ ).

4787 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
4788 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$4789 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4790 \\ 4791 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}),$$

4792 where the difference operator refers to set difference.

- 4793 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$4794 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4795 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
4796 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$4797 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 4798 \\ 4799 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4800 \\ 4801 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4802 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4803 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
4804 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
4805 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

4839 bool or any of the predefined “built-in” types in Table 3.2. If the default mask  
 4840 is desired (i.e., a mask that is all true with the dimensions of a column of C),  
 4841 GrB\_NULL should be specified.

4842 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
 4843 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
 4844 specified.

4845 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a column  
 4846 of C.

4847 **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
 4848 the specified column of C that are to be assigned. If all elements of the column  
 4849 in C are to be assigned in order from index 0 to **nrows** – 1, then GrB\_ALL should  
 4850 be specified. Regardless of execution mode and return value, this array may be  
 4851 manipulated by the caller after this operation returns without affecting any de-  
 4852 ferred computations for this operation. If this array contains duplicate values, it  
 4853 implies in assignment of more than one value to the same location which leads to  
 4854 undefined results.

4855 **nrows** (IN) The number of values in **row\_indices** array. Must be equal to **size(u)**.

4856 **col\_index** (IN) The index of the column in C to assign. Must be in the range [0, **ncols(C)**).

4857 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 4858 should be specified. Non-default field/value pairs are listed as follows:

4859

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.

4860

## 4861 Return Values

4862 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 4863 blocking mode, this indicates that the compatibility tests on  
 4864 dimensions and domains for the input arguments passed suc-  
 4865 cessfully. Either way, output matrix C is ready to be used in the  
 4866 next method of the sequence.

4867 **GrB\_PANIC** Unknown internal error.

4868           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
4869           opaque GraphBLAS objects (input or output) is in an invalid  
4870           state caused by a previous execution error. Call `GrB_error()` to  
4871           access any error messages generated by the implementation.

4872           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

4873           GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
4874           by a call to `new` (or `dup` for vector or matrix parameters).

4875           GrB\_INVALID\_INDEX `col_index` is outside the allowable range (i.e., greater than `ncols(C)`).

4876           GrB\_INDEX\_OUT\_OF\_BOUNDS A value in `row_indices` is greater than or equal to `nrows(C)`. In  
4877           non-blocking mode, this can be reported as an execution error.

4878           GrB\_DIMENSION\_MISMATCH `mask` size and number of rows in `C` are not the same, or `nrows`  $\neq$   
4879           `size(u)`.

4880           GrB\_DOMAIN\_MISMATCH The domains of the matrix and vector are incompatible with  
4881           each other or the corresponding domains of the accumulation  
4882           operator, or the mask's domain is not compatible with `bool` (in  
4883           the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

4884           GrB\_NULL\_POINTER Argument `row_indices` is a NULL pointer.

## 4885 Description

4886 This variant of `GrB_assign` computes the result of assigning a subset of locations in a column of a  
4887 GraphBLAS matrix (in a specific order) from the contents of a GraphBLAS vector:  
4888  $C(:, col\_index) = u$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C(:, col\_index) =$   
4889  $C(:, col\_index) \odot u$ . Taking order of `row_indices` into account, it is more explicitly written as:

$$4890 \quad C(row\_indices[i], col\_index) = u(i), \forall i : 0 \leq i < nrows, \text{ or}$$

$$C(row\_indices[i], col\_index) = C(row\_indices[i], col\_index) \odot u(i), \forall i : 0 \leq i < nrows.$$

4891 Logically, this operation occurs in three steps:

4892       **Setup** The internal matrices, vectors and mask used in the computation are formed and their  
4893       domains and dimensions are tested for compatibility.

4894       **Compute** The indicated computations are carried out.

4895       **Output** The result is written into the output matrix, possibly under control of a mask.

4896 Up to three argument vectors and matrices are used in this `GrB_assign` operation:

- 4897 1.  $C = \langle D(C), nrows(C), ncols(C), L(C) = \{(i, j, C_{ij})\} \rangle$
- 4898 2.  $mask = \langle D(mask), size(mask), L(mask) = \{(i, m_i)\} \rangle$  (optional)

4899 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4900 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain  
4901 compatibility as follows:

- 4902 1. If **mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\mathbf{mask})$   
4903 must be from one of the pre-defined types of Table 3.2.
- 4904 2.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .
- 4905 3. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
4906 of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accu-  
4907 mulation operator.

4908 Two domains are compatible with each other if values from one domain can be cast to values in  
4909 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4910 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4911 any compatibility rule above is violated, execution of **GrB\_assign** ends and the domain mismatch  
4912 error listed above is returned.

4913 The **col\_index** parameter is checked for a valid value. The following condition must hold:

- 4914 1.  $0 \leq \mathbf{col\_index} < \mathbf{ncols}(\mathbf{C})$

4915 If the rule above is violated, execution of **GrB\_assign** ends and the invalid index error listed above  
4916 is returned.

4917 From the arguments, the internal vectors, **mask**, and index array used in the computation are  
4918 formed ( $\leftarrow$  denotes copy):

- 4919 1. The vector,  $\tilde{\mathbf{c}}$ , is extracted from a column of **C** as follows:

$$4920 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \{(i, C_{ij}) \mid i : 0 \leq i < \mathbf{nrows}(\mathbf{C}), j = \mathbf{col\_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

- 4921 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument **mask** as follows:

- 4922 (a) If **mask** = **GrB\_NULL**, then  $\tilde{\mathbf{m}} = \langle \mathbf{nrows}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C})\} \rangle$ .
- 4923 (b) If **mask**  $\neq$  **GrB\_NULL**,
  - 4924 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
  - 4925 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
- 4926 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .

- 4927 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

- 4928 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument **row\_indices** as follows:

- 4929 (a) If **row\_indices** = **GrB\_ALL**, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .



4930 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{row\_indices}[i]$ ,  $\forall i : 0 \leq i < \text{nrows}$ .

4931 The internal vectors, matrices, and masks are checked for dimension compatibility. The following  
4932 conditions must hold:

- 4933 1.  $\text{size}(\tilde{\mathbf{c}}) = \text{size}(\tilde{\mathbf{m}})$
- 4934 2.  $\text{nrows} = \text{size}(\tilde{\mathbf{u}})$ .

4935 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
4936 match error listed above is returned.

4937 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4938 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4939 We are now ready to carry out the assign and any additional associated operations. We describe  
4940 this in terms of two intermediate vectors:

- 4941 •  $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{c}}$ .
- 4942 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4943 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$4944 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nrows} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4945 At this point, if any value of  $\tilde{\mathbf{I}}[i]$  is outside the valid range of indices for vector  $\tilde{\mathbf{c}}$ , computation  
4946 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`  
4947 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
4948 result matrix,  $\mathbf{C}$ , is invalid from this point forward in the sequence.

4949 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

- 4950 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$4951 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4952 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
4953 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  
4954 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\text{ind}(\tilde{\mathbf{t}})$ ).

4955 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
4956 indices in  $\tilde{\mathbf{c}}$  and  $\tilde{\mathbf{t}}$ .

$$4957 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))),$$

$$4958 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4960 where the difference operator refers to set difference.

4961 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$4962 \quad \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4963 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
4964 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$4965 \quad z_i = \tilde{\mathbf{c}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$4966 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$4967 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

4970 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4971 Finally, the set of output values that make up the  $\tilde{\mathbf{z}}$  vector are written into the column of the final  
4972 result matrix,  $\mathbf{C}(:, \text{col\_index})$ . This is carried out under control of the mask which acts as a “write  
4973 mask”.

4974 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}(:, \text{col\_index})$  on input to this  
4975 operation are deleted and the new contents of the column is given by:

$$4976 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : j \neq \text{col\_index}\} \cup \{(i, \text{col\_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

4977 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
4978 copied into the column of the final result matrix,  $\mathbf{C}(:, \text{col\_index})$ , and elements of this column  
4979 that fall outside the set indicated by the mask are unchanged:

$$4980 \quad \begin{aligned} \mathbf{L}(\mathbf{C}) = & \{(i, j, C_{ij}) : j \neq \text{col\_index}\} \cup \\ 4981 & \{(i, \text{col\_index}, \tilde{\mathbf{c}}(i)) : i \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \\ 4982 & \{(i, \text{col\_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}. \end{aligned}$$

4983 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
4984 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
4985 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but may  
4986 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 4987 4.3.7.4 assign: Row variant

4988 Assign the contents a vector to a subset of elements in one row of a matrix. Note that since the  
4989 output cannot be transposed, a different variant of `assign` is provided to assign to a column of a  
4990 matrix.

## 4991 C Syntax

```

4992         GrB_Info GrB_assign(GrB_Matrix      C,
4993                             const GrB_Vector mask,
4994                             const GrB_BinaryOp accum,
4995                             const GrB_Vector u,
4996                             GrB_Index      row_index,
4997                             const GrB_Index *col_indices,
4998                             GrB_Index      ncols,
4999                             const GrB_Descriptor desc);

```

## 5000 Parameters

5001       **C** (INOUT) An existing GraphBLAS Matrix. On input, the matrix provides values  
5002       that may be accumulated with the result of the assign operation. On output, this  
5003       matrix holds the results of the operation.

5004       **mask** (IN) An optional “write” mask that controls which results from this operation are  
5005       stored into the specified row of the output matrix **C**. The mask dimensions must  
5006       match those of a single row of the matrix **C**. If the **GrB\_STRUCTURE** descriptor  
5007       is *not* set for the mask, the domain of the **Mask** matrix must be of type **bool** or  
5008       any of the predefined “built-in” types in Table 3.2. If the default mask is desired  
5009       (i.e., a mask that is all **true** with the dimensions of a row of **C**), **GrB\_NULL** should  
5010       be specified.

5011       **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
5012       entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5013       specified.

5014       **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a row of  
5015       **C**.

5016       **row\_index** (IN) The index of the row in **C** to assign. Must be in the range  $[0, \mathbf{nrows}(\mathbf{C})]$ .

5017       **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
5018       the specified row of **C** that are to be assigned. If all elements of the row in **C** are to  
5019       be assigned in order from index 0 to  $\mathbf{ncols} - 1$ , then **GrB\_ALL** should be specified.  
5020       Regardless of execution mode and return value, this array may be manipulated by  
5021       the caller after this operation returns without affecting any deferred computations  
5022       for this operation. If this array contains duplicate values, it implies in assignment  
5023       of more than one value to the same location which leads to undefined results.

5024       **ncols** (IN) The number of values in **col\_indices** array. Must be equal to **size(u)**.

5025       **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5026       should be specified. Non-default field/value pairs are listed as follows:

5027

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output row in C is cleared (all elements removed) before result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of <b>mask</b> .

## Return Values

GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.
GrB_PANIC	Unknown internal error.
GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call <b>GrB_error()</b> to access any error messages generated by the implementation.
GrB_OUT_OF_MEMORY	Not enough memory available for operation.
GrB_UNINITIALIZED_OBJECT	One or more of the GraphBLAS objects has not been initialized by a call to <b>new</b> (or <b>dup</b> for vector or matrix parameters).
GrB_INVALID_INDEX	<b>row_index</b> is outside the allowable range (i.e., greater than <b>nrows(C)</b> ).
GrB_INDEX_OUT_OF_BOUNDS	A value in <b>col_indices</b> is greater than or equal to <b>ncols(C)</b> . In non-blocking mode, this can be reported as an execution error.
GrB_DIMENSION_MISMATCH	<b>mask</b> size and number of columns in C are not the same, or <b>ncols</b> $\neq$ <b>size(u)</b> .
GrB_DOMAIN_MISMATCH	The domains of the matrix and vector are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with <b>bool</b> (in the case where <b>desc[GrB_MASK].GrB_STRUCTURE</b> is not set).
GrB_NULL_POINTER	Argument <b>col_indices</b> is a NULL pointer.

## Description

This variant of **GrB\_assign** computes the result of assigning a subset of locations in a row of a GraphBLAS matrix (in a specific order) from the contents of a GraphBLAS vector:

5056  $C(\text{row\_index}, :) = u$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C(\text{row\_index}, :$   
 5057  $) = C(\text{row\_index}, :) \odot u$ . Taking order of `col_indices` into account it is more explicitly written as:

$$5058 \quad C(\text{row\_index}, \text{col\_indices}[j]) = u(j), \forall j : 0 \leq j < \text{ncols}, \text{ or}$$

$$5058 \quad C(\text{row\_index}, \text{col\_indices}[j]) = C(\text{row\_index}, \text{col\_indices}[j]) \odot u(j), \forall j : 0 \leq j < \text{ncols}$$

5059 Logically, this operation occurs in three steps:

5060     **Setup** The internal matrices, vectors and mask used in the computation are formed and their  
 5061             domains and dimensions are tested for compatibility.

5062     **Compute** The indicated computations are carried out.

5063     **Output** The result is written into the output matrix, possibly under control of a mask.

5064 Up to three argument vectors and matrices are used in this `GrB_assign` operation:

- 5065     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5066     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 5067     3.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5068 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain  
 5069 compatibility as follows:

- 5070     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
 5071         must be from one of the pre-defined types of Table 3.2.
- 5072     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(u)$ .
- 5073     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 5074         of the accumulation operator and  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 5075         mulation operator.

5076 Two domains are compatible with each other if values from one domain can be cast to values in  
 5077 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 5078 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 5079 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
 5080 error listed above is returned.

5081 The `row_index` parameter is checked for a valid value. The following condition must hold:

- 5082     1.  $0 \leq \text{row\_index} < \mathbf{nrows}(C)$

5083 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above  
 5084 is returned.

5085 From the arguments, the internal vectors, mask, and index array used in the computation are  
 5086 formed ( $\leftarrow$  denotes copy):

5087 1. The vector,  $\tilde{\mathbf{c}}$ , is extracted from a row of  $\mathbf{C}$  as follows:

$$5088 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(j, C_{ij}) \mid \forall j : 0 \leq j < \mathbf{ncols}(\mathbf{C}), i = \text{row\_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

5089 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:

5090 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{ncols}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{ncols}(\mathbf{C})\} \rangle$ .

5091 (b) If `mask  $\neq$  GrB_NULL`,

5092 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,

5093 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .

5094 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .

5095 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

5096 4. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:

5097 (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$ .

5098 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \text{col\_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$ .

5099 The internal vectors, matrices, and masks are checked for dimension compatibility. The following  
5100 conditions must hold:

5101 1.  $\mathbf{size}(\tilde{\mathbf{c}}) = \mathbf{size}(\tilde{\mathbf{m}})$

5102 2.  $\mathbf{ncols} = \mathbf{size}(\tilde{\mathbf{u}})$ .

5103 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
5104 match error listed above is returned.

5105 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
5106 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5107 We are now ready to carry out the assign and any additional associated operations. We describe  
5108 this in terms of two intermediate vectors:

- 5109 •  $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{c}}$ .
- 5110 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

5111 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$5112 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{J}}[j], \tilde{\mathbf{u}}(j)) \mid \forall j, 0 \leq j < \mathbf{ncols} : j \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle.$$

5113 At this point, if any value of  $\tilde{\mathbf{J}}[j]$  is outside the valid range of indices for vector  $\tilde{\mathbf{c}}$ , computation  
5114 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`  
5115 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
5116 result matrix,  $\mathbf{C}$ , is invalid from this point forward in the sequence.

5117 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

5118 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$5119 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5120 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
5121 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  
5122 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\mathbf{ind}(\tilde{\mathbf{t}})$ ).

5123 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
5124 indices in  $\tilde{\mathbf{c}}$  and  $\tilde{\mathbf{t}}$ .

$$5125 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5126 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5127 where the difference operator refers to set difference.

5128 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$5129 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{c}}), \{(j, z_j) \mid j \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5130 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
5131 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$5132 \quad z_j = \tilde{\mathbf{c}}(j) \odot \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$5133 \quad z_j = \tilde{\mathbf{c}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5134 \quad z_j = \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

5135 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

5136 Finally, the set of output values that make up the  $\tilde{\mathbf{z}}$  vector are written into the column of the final  
5137 result matrix,  $\mathbf{C}(\text{row\_index}, :)$ . This is carried out under control of the mask which acts as a “write  
5138 mask”.

5139 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}(\text{row\_index}, :)$  on input to this  
5140 operation are deleted and the new contents of the column is given by:

$$5141 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row\_index}\} \cup \{(\text{row\_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5142 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
5143 copied into the column of the final result matrix,  $\mathbf{C}(\text{row\_index}, :)$ , and elements of this column  
5144 that fall outside the set indicated by the mask are unchanged:

$$5145 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row\_index}\} \cup$$

$$5146 \quad \{(\text{row\_index}, j, \tilde{\mathbf{c}}(j)) : j \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup$$

$$5147 \quad \{(\text{row\_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5148 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
5149 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
5150 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but may  
5151 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

5155 **4.3.7.5 assign: Constant vector variant**[Scott: NEW CONTENT]

5156 Assign the same value to a specified subset of vector elements. With the use of GrB\_ALL, the entire  
5157 destination vector can be filled with the constant.

5158 **C Syntax**

```
5159         GrB_Info GrB_assign(GrB_Vector          w,  
5160                             const GrB_Vector    mask,  
5161                             const GrB_BinaryOp   accum,  
5162                             <type>              val,  
5163                             const GrB_Index      *indices,  
5164                             GrB_Index            nindices,  
5165                             const GrB_Descriptor desc);
```

```
5166         GrB_Info GrB_assign(GrB_Vector          w,  
5167                             const GrB_Vector    mask,  
5168                             const GrB_BinaryOp   accum,  
5169                             const GrB_Scalar     s,  
5170                             const GrB_Index      *indices,  
5171                             GrB_Index            nindices,  
5172                             const GrB_Descriptor desc);
```

5173 **Parameters**

5174 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
5175 that may be accumulated with the result of the assign operation. On output, this  
5176 vector holds the results of the operation.

5177 **mask** (IN) An optional “write” mask that controls which results from this operation are  
5178 stored into the output vector w. The mask dimensions must match those of the  
5179 vector w. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
5180 of the mask vector must be of type bool or any of the predefined “built-in” types  
5181 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
5182 dimensions of w), GrB\_NULL should be specified.

5183 **accum** (IN) An optional binary operator used for accumulating entries into existing w  
5184 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
5185 specified.

5186 **val** (IN) Scalar value to assign to (a subset of) w.

5187 **s** (IN) Scalar value to assign to (a subset of) w.

5188 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
5189 w that are to be assigned. If all elements of w are to be assigned in order from 0



5190 to `nindices - 1`, then `GrB_ALL` should be specified. Regardless of execution mode  
5191 and return value, this array may be manipulated by the caller after this operation  
5192 returns without affecting any deferred computations for this operation. In this  
5193 variant, the specific order of the values in the array has no effect on the result.  
5194 Unlike other variants, if there are duplicated values in this array the result is still  
5195 defined.

5196 **nindices** (IN) The number of values in `indices` array. Must be in the range: `[0, size(w)]`. If  
5197 `nindices` is zero, the operation becomes a NO-OP.

5198 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
5199 should be specified. Non-default field/value pairs are listed as follows:

5200

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> .

5201

## 5202 Return Values

5203 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
5204 blocking mode, this indicates that the compatibility tests on  
5205 dimensions and domains for the input arguments passed suc-  
5206 cessfully. Either way, output vector `w` is ready to be used in the  
5207 next method of the sequence.

5208 **GrB\_PANIC** Unknown internal error.

5209 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
5210 opaque GraphBLAS objects (input or output) is in an invalid  
5211 state caused by a previous execution error. Call `GrB_error()` to  
5212 access any error messages generated by the implementation.

5213 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

5214 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized  
5215 by a call to `new` (or `dup` for vector parameters).

5216 **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in `indices` is greater than or equal to `size(w)`. In non-  
5217 blocking mode, this can be reported as an execution error.

5218 **GrB\_DIMENSION\_MISMATCH** `mask` and `w` dimensions are incompatible, or `nindices` is not less  
5219 than `size(w)`.



5250 4. If **accum** is not **GrB\_NULL**, then either **D(val)** or **D(s)**, depending on the signature of the  
 5251 method, must be compatible with **D<sub>in2</sub>(accum)** of the accumulation operator.

5252 Two domains are compatible with each other if values from one domain can be cast to values in  
 5253 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 5254 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 5255 any compatibility rule above is violated, execution of **GrB\_assign** ends and the domain mismatch  
 5256 error listed above is returned.

5257 From the arguments, the internal vectors, mask and index array used in the computation are formed  
 5258 ( $\leftarrow$  denotes copy):

- 5259 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5260 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument **mask** as follows:
  - 5261 (a) If **mask** = **GrB\_NULL**, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 5262 (b) If **mask**  $\neq$  **GrB\_NULL**,
    - 5263 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 5264 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}(\mathbf{mask})(i) = \mathbf{true})\} \rangle$ .
  - 5265 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 5266 3. Scalar  $\tilde{s} \leftarrow s$  (**GrB\_Scalar** version only).
- 5267 4. The internal index array,  $\tilde{\mathbf{I}}$ , is computed from argument **indices** as follows:
  - 5268 (a) If **indices** = **GrB\_ALL**, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nindices}$ .
  - 5269 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{indices}[i], \forall i : 0 \leq i < \mathbf{nindices}$ .

5270 The internal vector and mask are checked for dimension compatibility. The following conditions  
 5271 must hold:

- 5272 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5273 2.  $0 \leq \mathbf{nindices} \leq \mathbf{size}(\tilde{\mathbf{w}})$ .

5274 If any compatibility rule above is violated, execution of **GrB\_assign** ends and the dimension mis-  
 5275 match error listed above is returned.

5276 From this point forward, in **GrB\_NONBLOCKING** mode, the method can optionally exit with  
 5277 **GrB\_SUCCESS** return code and defer any computation and/or execution error codes.

5278 We are now ready to carry out the assign and any additional associated operations. We describe  
 5279 this in terms of two intermediate vectors:

- 5280 •  $\tilde{\mathbf{t}}$ : The vector holding the copies of the scalar, either **val** or  $\tilde{s}$ , in their destination locations  
 5281 relative to  $\tilde{\mathbf{w}}$ .

- $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows. If a non-opaque scalar  $\mathbf{val}$  is provided:

$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{val}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

Correspondingly, if a non-empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 1$ ):

$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}(\tilde{s})) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

Finally, if an empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 0$ ):

$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \emptyset \rangle.$$

If  $\tilde{\mathbf{I}}$  is empty, this operation results in an empty vector,  $\tilde{\mathbf{t}}$ . Otherwise, if any value in the  $\tilde{\mathbf{I}}$  array is not in the range  $[0, \mathbf{size}(\tilde{\mathbf{w}}))$ , the execution of `GrB_assign` ends and the index out-of-bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the result vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

- If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$\tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure of  $\tilde{\mathbf{w}}$  ( $\mathbf{ind}(\tilde{\mathbf{w}})$ ) and remove from it all the indices of  $\tilde{\mathbf{w}}$  that are in the set of indices being assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\mathbf{ind}(\tilde{\mathbf{t}})$ ).

The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

where the difference operator refers to set difference. We note that in this case of assigning a constant,  $\{\tilde{\mathbf{I}}[k], \forall k\}$  and  $\mathbf{ind}(\tilde{\mathbf{t}})$  are identical.

- If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$\langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

5317 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 5318 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 5319 mask which acts as a “write mask”.

- 5320 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are  
 5321 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$5322 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 5323 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 5324 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 5325 mask are unchanged:

$$5326 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5327 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 5328 of vector  $\mathbf{w}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 5329 exits with return value GrB\_SUCCESS and the new content of vector  $\mathbf{w}$  is as defined above but  
 5330 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 5331 sequence.

#### 5332 4.3.7.6 assign: Constant matrix variant[Scott: NEW CONTENT]

5333 Assign the same value to a specified subset of matrix elements. With the use of GrB\_ALL, the  
 5334 entire destination matrix can be filled with the constant.

### 5335 C Syntax

```
5336      GrB_Info GrB_assign(GrB_Matrix      C,
5337                        const GrB_Matrix  Mask,
5338                        const GrB_BinaryOp accum,
5339                        <type>            val,
5340                        const GrB_Index    *row_indices,
5341                        GrB_Index          nrows,
5342                        const GrB_Index    *col_indices,
5343                        GrB_Index          ncols,
5344                        const GrB_Descriptor desc);
```

```
5345      GrB_Info GrB_assign(GrB_Matrix      C,
5346                        const GrB_Matrix  Mask,
5347                        const GrB_BinaryOp accum,
5348                        const GrB_Scalar   s,
5349                        const GrB_Index    *row_indices,
5350                        GrB_Index          nrows,
```

```

5351         const GrB_Index      *col_indices,
5352         GrB_Index             ncols,
5353         const GrB_Descriptor  desc);

```

## 5354 Parameters

5355 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
5356 that may be accumulated with the result of the assign operation. On output, the  
5357 matrix holds the results of the operation.

5358 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
5359 stored into the output matrix **C**. The mask dimensions must match those of the  
5360 matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5361 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
5362 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5363 dimensions of **C**), **GrB\_NULL** should be specified.

5364 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
5365 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5366 specified.

5367 **val** (IN) Scalar value to assign to (a subset of) **C**.

5368 **s** (IN) Scalar value to assign to (a subset of) **C**.

5369 **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**  
5370 that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** − 1,  
5371 then **GrB\_ALL** can be specified. Regardless of execution mode and return value,  
5372 this array may be manipulated by the caller after this operation returns without  
5373 affecting any deferred computations for this operation. Unlike other variants, if  
5374 there are duplicated values in this array the result is still defined.

5375 **nrows** (IN) The number of values in **row\_indices** array. Must be in the range: [0, **nrows**(**C**)].  
5376 If **nrows** is zero, the operation becomes a NO-OP.

5377 **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns of **C**  
5378 that are assigned. If all columns of **C** are to be assigned in order from 0 to **ncols** − 1,  
5379 then **GrB\_ALL** should be specified. Regardless of execution mode and return value,  
5380 this array may be manipulated by the caller after this operation returns without  
5381 affecting any deferred computations for this operation. Unlike other variants, if  
5382 there are duplicated values in this array the result is still defined.

5383 **ncols** (IN) The number of values in **col\_indices** array. Must be in the range: [0, **ncols**(**C**)].  
5384 If **ncols** is zero, the operation becomes a NO-OP.

5385 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5386 should be specified. Non-default field/value pairs are listed as follows:

5387

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`

5417 or  $C(\text{row\_indices}, \text{col\_indices}) = s$  is performed. If an optional binary accumulation operator  
 5418  $(\odot)$  is provided, then either  $C(\text{row\_indices}, \text{col\_indices}) = C(\text{row\_indices}, \text{col\_indices}) \odot \text{val}$  or  
 5419  $C(\text{row\_indices}, \text{col\_indices}) = C(\text{row\_indices}, \text{col\_indices}) \odot s$  is performed. More explicitly, if a  
 5420 non-opaque value  $\text{val}$  is provided:

$$\begin{aligned} & C(\text{row\_indices}[i], \text{col\_indices}[j]) = \text{val}, \text{ or} \\ 5421 \quad & C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot \text{val} \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5422 Correspondingly, if a `GrB_Scalar`  $s$  is provided:

$$\begin{aligned} & C(\text{row\_indices}[i], \text{col\_indices}[j]) = s, \text{ or} \\ 5423 \quad & C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot s \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5424 Logically, this operation occurs in three steps:

5425     Setup The internal vectors and mask used in the computation are formed and their domains  
 5426             and dimensions are tested for compatibility.

5427     Compute The indicated computations are carried out.

5428     Output The result is written into the output matrix, possibly under control of a mask.

5429 Up to two argument matrices are used in the `GrB_assign` operation:

- 5430     1.  $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5431     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)

5432 The argument scalar, matrices, and the accumulation operator (if provided) are tested for domain  
 5433 compatibility as follows:

- 5434     1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 5435         must be from one of the pre-defined types of Table 3.2.
- 5436     2.  $\mathbf{D}(C)$  must be compatible with either  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(s)$ , depending on the signature of the  
 5437         method.
- 5438     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 5439         of the accumulation operator.
- 5440     4. If `accum` is not `GrB_NULL`, then either  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(s)$ , depending on the signature of the  
 5441         method, must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.



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

From the arguments, the internal matrices, index arrays, and mask used in the computation are formed ( $\leftarrow$  denotes copy):

1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
2. Two-dimensional mask  $\tilde{\mathbf{M}}$  is computed from argument `Mask` as follows:
  - (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - (b) If `Mask  $\neq$  GrB_NULL`,
    - i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
3. Scalar  $\tilde{s} \leftarrow s$  (`GrB_Scalar` version only).
4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
  - (a) If `row_indices = GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .
  - (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{row\_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .
5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:
  - (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$ .
  - (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \mathbf{col\_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$ .

The internal matrix and mask are checked for dimension compatibility. The following conditions must hold:

1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
3.  $0 \leq \mathbf{nrows} \leq \mathbf{nrows}(\tilde{\mathbf{C}})$ .
4.  $0 \leq \mathbf{ncols} \leq \mathbf{ncols}(\tilde{\mathbf{C}})$ .

If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mismatch error listed above is returned.

5474 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 5475 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5476 We are now ready to carry out the assign and any additional associated operations. We describe  
 5477 this in terms of two intermediate matrices:

- 5478 •  $\tilde{\mathbf{T}}$ : The matrix holding the copies of the scalar, either `val` or  $\tilde{s}$ , in their destination locations  
 5479 relative to  $\tilde{\mathbf{C}}$ .
- 5480 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

5481 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows. If a non-opaque scalar `val` is provided:

$$5482 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\text{val}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle.$$

5483 Correspondingly, if a non-empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e., `size`( $\tilde{s}$ ) = 1):

$$5484 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}(\tilde{s})) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle.$$

5485 Finally, if an empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e., `size`( $\tilde{s}$ ) = 0):

$$5486 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \emptyset \rangle.$$

5487 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  
 5488 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  
 5489  $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$ , the execution of `GrB_assign` ends and the index out-of-bounds error listed above is  
 5490 generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating  
 5491 `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from this point forward in the  
 5492 sequence.

5493 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows:

- 5494 • If `accum` = `GrB_NULL`, then  $\tilde{\mathbf{Z}}$  is defined as

$$5495 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 5496 \quad \{(i, j, Z_{ij}) \mid (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}}))\} \rangle.$$

5497 The above expression defines the structure of matrix  $\tilde{\mathbf{Z}}$  as follows: We start with the structure  
 5498 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  
 5499 assigned ( $\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$ ). Finally, we add the structure of  $\tilde{\mathbf{T}}$  ( $\mathbf{ind}(\tilde{\mathbf{T}})$ ).

5500 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 5501 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$5502 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5503 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}), \\ 5504$$

5505 where the difference operator refers to set difference. We note that, in this particular case of  
 5506 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.

5507 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$5508 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5509 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
5510 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$5511 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$5512 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$5513 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5516 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

5517 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
5518 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
5519 mask which acts as a “write mask”.

5520 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
5521 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$5522 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5523 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
5524 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
5525 mask are unchanged:

$$5526 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5527 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
5528 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
5529 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
5530 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
5531 sequence.

### 5532 4.3.8 apply: Apply a function to the elements of an object

5533 Computes the transformation of the values of the elements of a vector or a matrix using a unary  
5534 function, or a binary function where one argument is bound to a scalar.

#### 5535 4.3.8.1 apply: Vector variant

5536 Computes the transformation of the values of the elements of a vector using a unary function.

## 5537 C Syntax

```

5538      GrB_Info GrB_apply(GrB_Vector      w,
5539                        const GrB_Vector  mask,
5540                        const GrB_BinaryOp accum,
5541                        const GrB_UnaryOp  op,
5542                        const GrB_Vector  u,
5543                        const GrB_Descriptor desc);

```

## 5544 Parameters

5545 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
 5546 that may be accumulated with the result of the apply operation. On output, this  
 5547 vector holds the results of the operation.

5548 **mask** (IN) An optional “write” mask that controls which results from this operation are  
 5549 stored into the output vector **w**. The mask dimensions must match those of the  
 5550 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
 5551 of the mask vector must be of type **bool** or any of the predefined “built-in” types  
 5552 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
 5553 dimensions of **w**), **GrB\_NULL** should be specified.

5554 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
 5555 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
 5556 specified.

5557 **op** (IN) A unary operator applied to each element of input vector **u**.

5558 **u** (IN) The GraphBLAS vector to which the unary function is applied.

5559 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
 5560 should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
<b>w</b>	<b>GrB_OUTP</b>	<b>GrB_REPLACE</b>	Output vector <b>w</b> is cleared (all elements removed) before the result is stored in it.
<b>mask</b>	<b>GrB_MASK</b>	<b>GrB_STRUCTURE</b>	The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined.
<b>mask</b>	<b>GrB_MASK</b>	<b>GrB_COMP</b>	Use the complement of <b>mask</b> .

## 5563 Return Values

5564 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 5565 blocking mode, this indicates that the compatibility tests on di-  
 5566 mensions and domains for the input arguments passed successfully.

5567                                Either way, output vector  $w$  is ready to be used in the next method  
5568                                of the sequence.

5569                                **GrB\_PANIC** Unknown internal error.

5570                                **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
5571                                GraphBLAS objects (input or output) is in an invalid state caused  
5572                                by a previous execution error. Call **GrB\_error()** to access any error  
5573                                messages generated by the implementation.

5574                                **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

5575 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
5576                                a call to **new** (or **dup** for vector parameters).

5577 **GrB\_DIMENSION\_MISMATCH**  $mask$ ,  $w$  and/or  $u$  dimensions are incompatible.

5578                                **GrB\_DOMAIN\_MISMATCH** The domains of the various vectors are incompatible with the corre-  
5579                                sponding domains of the accumulation operator or unary function,  
5580                                or the mask's domain is not compatible with **bool** (in the case where  
5581                                 $desc[GrB\_MASK].GrB\_STRUCTURE$  is not set).

## 5582 **Description**

5583    This variant of **GrB\_apply** computes the result of applying a unary function to the elements of a  
5584    GraphBLAS vector:  $w = f(u)$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  
5585     $w = w \odot f(u)$ .

5586    Logically, this operation occurs in three steps:

5587                                **Setup** The internal vectors and mask used in the computation are formed and their domains  
5588                                and dimensions are tested for compatibility.

5589                                **Compute** The indicated computations are carried out.

5590                                **Output** The result is written into the output vector, possibly under control of a mask.

5591    Up to three argument vectors are used in this **GrB\_apply** operation:

- 5592        1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 5593        2.  $mask = \langle \mathbf{D}(mask), \mathbf{size}(mask), \mathbf{L}(mask) = \{(i, m_i)\} \rangle$  (optional)
- 5594        3.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5595    The argument vectors, unary operator and the accumulation operator (if provided) are tested for  
5596    domain compatibility as follows:

- 5597 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
5598 must be from one of the pre-defined types of Table 3.2.
- 5599 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the unary operator.
- 5600 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
5601 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the unary operator must be compatible with  
5602  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 5603 4.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in}(\text{op})$ .

5604 Two domains are compatible with each other if values from one domain can be cast to values in  
5605 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
5606 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
5607 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
5608 error listed above is returned.

5609 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
5610 denotes copy):

- 5611 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5612 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 5613 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .
  - 5614 (b) If `mask`  $\neq$  `GrB_NULL`,
    - 5615 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
    - 5616 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 5617 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 5618 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

5619 The internal vectors and masks are checked for dimension compatibility. The following conditions  
5620 must hold:

- 5621 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$
- 5622 2.  $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$ .

5623 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
5624 error listed above is returned.

5625 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
5626 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5627 We are now ready to carry out the apply and any additional associated operations. We describe  
5628 this in terms of two intermediate vectors:

- $\tilde{\mathbf{t}}$ : The vector holding the result from applying the unary operator to the input vector  $\tilde{\mathbf{u}}$ .
- $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f(\tilde{\mathbf{u}}(i))) \mid \forall i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle,$$

where  $f = \mathbf{f}(\text{op})$ .

The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ , using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a “write mask”.

- If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

- If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

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

### 4.3.8.2 apply: Matrix variant

Computes the transformation of the values of the elements of a matrix using a unary function.

#### C Syntax

```
GrB_Info GrB_apply(GrB_Matrix      C,
                  const GrB_Matrix  Mask,
                  const GrB_BinaryOp accum,
                  const GrB_UnaryOp  op,
                  const GrB_Matrix  A,
                  const GrB_Descriptor desc);
```

#### Parameters

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

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

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

**op** (IN) A unary operator applied to each element of input matrix A.

**A** (IN) The GraphBLAS matrix to which the unary function is applied.

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

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input <b>Mask</b> matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of <b>Mask</b> .
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.



## 5689 Return Values

5690                   GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
5691                   blocking mode, this indicates that the compatibility tests on  
5692                   dimensions and domains for the input arguments passed suc-  
5693                   cessfully. Either way, output matrix C is ready to be used in the  
5694                   next method of the sequence.

5695                   GrB\_PANIC Unknown internal error.

5696                   GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
5697                   opaque GraphBLAS objects (input or output) is in an invalid  
5698                   state caused by a previous execution error. Call `GrB_error()` to  
5699                   access any error messages generated by the implementation.

5700                   GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

5701                   GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
5702                   by a call to `new` (or `Matrix_dup` for matrix parameters).

5703                   GrB\_DIMENSION\_MISMATCH Mask and C dimensions are incompatible,  $\mathbf{nrows} \neq \mathbf{nrows}(C)$ , or  
5704                    $\mathbf{ncols} \neq \mathbf{ncols}(C)$ .

5705                   GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
5706                   corresponding domains of the accumulation operator or unary  
5707                   function, or the mask's domain is not compatible with `bool` (in  
5708                   the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 5709 Description

5710 This variant of `GrB_apply` computes the result of applying a unary function to the elements of a  
5711 GraphBLAS matrix:  $C = f(A)$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  
5712  $C = C \odot f(A)$ .

5713 Logically, this operation occurs in three steps:

5714       **Setup** The internal matrices and mask used in the computation are formed and their domains  
5715       and dimensions are tested for compatibility.

5716       **Compute** The indicated computations are carried out.

5717       **Output** The result is written into the output matrix, possibly under control of a mask.

5718 Up to three argument matrices are used in the `GrB_apply` operation:

- 5719       1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$   
5720       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)

5721 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

5722 The argument matrices, unary operator and the accumulation operator (if provided) are tested for  
5723 domain compatibility as follows:

- 5724 1. If **Mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\text{Mask})$   
5725 must be from one of the pre-defined types of Table 3.2.
- 5726 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the unary operator.
- 5727 3. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
5728 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the unary operator must be compatible with  
5729  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 5730 4.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in}(\text{op})$  of the unary operator.

5731 Two domains are compatible with each other if values from one domain can be cast to values in  
5732 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
5733 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
5734 any compatibility rule above is violated, execution of **GrB\_apply** ends and the domain mismatch  
5735 error listed above is returned.

5736 From the argument matrices, the internal matrices, mask, and index arrays used in the computation  
5737 are formed ( $\leftarrow$  denotes copy):

- 5738 1. Matrix  $\tilde{C} \leftarrow C$ .
- 5739 2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument **Mask** as follows:
  - 5740 (a) If **Mask** = **GrB\_NULL**, then  $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$   
5741  $j < \mathbf{ncols}(C)\} \rangle$ .
  - 5742 (b) If **Mask**  $\neq$  **GrB\_NULL**,
    - 5743 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
5744  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
    - 5745 ii. Otherwise,  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
5746  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
  - 5747 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\tilde{M} \leftarrow \neg \tilde{M}$ .
- 5748 3. Matrix  $\tilde{A} \leftarrow \text{desc[GrB_INP0].GrB_TRAN} ? A^T : A$ .

5749 The internal matrices and mask are checked for dimension compatibility. The following conditions  
5750 must hold:

- 5751 1.  $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{M})$ .
- 5752 2.  $\mathbf{ncols}(\tilde{C}) = \mathbf{ncols}(\tilde{M})$ .
- 5753 3.  $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{A})$ .

5754 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

5755 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
5756 error listed above is returned.

5757 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
5758 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5759 We are now ready to carry out the apply and any additional associated operations. We describe  
5760 this in terms of two intermediate matrices:

- 5761 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the unary operator to the input matrix  $\tilde{\mathbf{A}}$ .
- 5762 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

5763 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$5764 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j))) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

5765 where  $f = \mathbf{f}(\mathbf{op})$ .

5766 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 5767 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 5768 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$5769 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5770 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
5771 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$5772 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$5773 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$5774 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5775 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

5778 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
5779 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
5780 mask which acts as a “write mask”.

- 5781 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
5782 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$5783 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 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

```

5801 // bind-first + scalar value
5802 GrB_Info GrB_apply(GrB_Vector          w,
5803                   const GrB_Vector      mask,
5804                   const GrB_BinaryOp     accum,
5805                   const GrB_BinaryOp     op,
5806                   <type>                 val,
5807                   const GrB_Vector      u,
5808                   const GrB_Descriptor   desc);

5809 // bind-first + GraphBLAS scalar
5810 GrB_Info GrB_apply(GrB_Vector          w,
5811                   const GrB_Vector      mask,
5812                   const GrB_BinaryOp     accum,
5813                   const GrB_BinaryOp     op,
5814                   const GrB_Scalar      s,
5815                   const GrB_Vector      u,
5816                   const GrB_Descriptor   desc);

5817 // bind-second + scalar value
5818 GrB_Info GrB_apply(GrB_Vector          w,
5819                   const GrB_Vector      mask,
```

```

5820         const GrB_BinaryOp      accum,
5821         const GrB_BinaryOp      op,
5822         const GrB_Vector        u,
5823         <type>                  val,
5824         const GrB_Descriptor    desc);

5825 // bind-second + GraphBLAS scalar
5826 GrB_Info GrB_apply(GrB_Vector      w,
5827                   const GrB_Vector mask,
5828                   const GrB_BinaryOp accum,
5829                   const GrB_BinaryOp op,
5830                   const GrB_Vector u,
5831                   const GrB_Scalar s,
5832                   const GrB_Descriptor desc);

```

## 5833 Parameters

5834     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
5835     that may be accumulated with the result of the apply operation. On output, this  
5836     vector holds the results of the operation.

5837     **mask** (IN) An optional “write” mask that controls which results from this operation are  
5838     stored into the output vector **w**. The mask dimensions must match those of the  
5839     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5840     of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
5841     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5842     dimensions of **w**), **GrB\_NULL** should be specified.

5843     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
5844     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5845     specified.

5846     **op** (IN) A binary operator applied to each element of input vector, **u**, and the scalar  
5847     value, **val**.

5848     **u** (IN) The GraphBLAS vector whose elements are passed to the binary operator as  
5849     the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)  
5850     argument in the *bind-second* variant.

5851     **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)  
5852     argument in the *bind-first* variant, or the right-hand (second) argument in the  
5853     *bind-second* variant.

5854     **s** (IN) A GraphBLAS scalar that is passed to the binary operator as the left-hand  
5855     (first) argument in the *bind-first* variant, or the right-hand (second) argument in  
5856     the *bind-second* variant. It must not be empty.

5857 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 5858 should be specified. Non-default field/value pairs are listed as follows:

5859

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector <b>w</b> is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of <b>mask</b> .

5860

## 5861 Return Values

5862 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 5863 blocking mode, this indicates that the compatibility tests on di-  
 5864 mensions and domains for the input arguments passed successfully.  
 5865 Either way, output vector **w** is ready to be used in the next method  
 5866 of the sequence.

5867 GrB\_PANIC Unknown internal error.

5868 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 5869 GraphBLAS objects (input or output) is in an invalid state caused  
 5870 by a previous execution error. Call GrB\_error() to access any error  
 5871 messages generated by the implementation.

5872 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

5873 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
 5874 a call to new (or dup for vector parameters).

5875 GrB\_DIMENSION\_MISMATCH **mask**, **w** and/or **u** dimensions are incompatible.

5876 GrB\_DOMAIN\_MISMATCH The domains of the various vectors and scalar are incompatible with  
 5877 the corresponding domains of the binary operator or accumulation  
 5878 operator, or the **mask**'s domain is not compatible with **bool** (in the  
 5879 case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

5880 GrB\_EMPTY\_OBJECT The GrB\_Scalar **s** used in the call is empty (**nvals(s) = 0**) and  
 5881 therefore a value cannot be passed to the binary operator.

## 5882 Description

5883 This variant of GrB\_apply computes the result of applying a binary operator to the elements of a  
 5884 GraphBLAS vector each composed with a scalar constant, either **val** or **s**:

5885                   bind-first:      $w = f(\text{val}, u)$  or  $w = f(s, u)$

5886                   bind-second:    $w = f(u, \text{val})$  or  $w = f(u, s)$ ,

5887 or if an optional binary accumulation operator ( $\odot$ ) is provided:

5888                   bind-first:      $w = w \odot f(\text{val}, u)$  or  $w = w \odot f(s, u)$

5889                   bind-second:    $w = w \odot f(u, \text{val})$  or  $w = w \odot f(u, s)$ .

5890 Logically, this operation occurs in three steps:

5891     **Setup** The internal vectors and mask used in the computation are formed and their domains  
5892             and dimensions are tested for compatibility.

5893     **Compute** The indicated computations are carried out.

5894     **Output** The result is written into the output vector, possibly under control of a mask.

5895 Up to three argument vectors are used in this GrB\_apply operation:

5896     1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$

5897     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)

5898     3.  $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5899 The argument scalar, vectors, binary operator and the accumulation operator (if provided) are  
5900 tested for domain compatibility as follows:

5901     1. If **mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{mask})$   
5902         must be from one of the pre-defined types of Table 3.2.

5903     2.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the binary operator.

5904     3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
5905         of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the binary operator must be compatible with  
5906          $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.

5907     4.  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.

5908     5. If bind-first:

5909         (a)  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the binary operator.

5910         (b) If the non-opaque scalar **val** is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$   
5911             of the binary operator.

5912         (c) If the GrB\_Scalar **s** is provided, then  $\mathbf{D}(s)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the  
5913             binary operator.

- 5914 6. If bind-second:
- 5915 (a)  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the binary operator.
- 5916 (b) If the non-opaque scalar  $\mathbf{val}$  is provided, then  $\mathbf{D}(\mathbf{val})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$
- 5917 of the binary operator.
- 5918 (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
- 5919 binary operator.

5920 Two domains are compatible with each other if values from one domain can be cast to values in

5921 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all

5922 compatible with each other. A domain from a user-defined type is only compatible with itself. If

5923 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch

5924 error listed above is returned.

5925 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$

5926 denotes copy):

- 5927 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5928 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
- 5929 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
- 5930 (b) If `mask  $\neq$  GrB_NULL`,
- 5931 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
- 5932 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
- 5933 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 5934 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 5935 4. Scalar  $\tilde{\mathbf{s}} \leftarrow \mathbf{s}$  (GraphBLAS scalar case).

5936 The internal vectors and masks are checked for dimension compatibility. The following conditions

5937 must hold:

- 5938 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5939 2.  $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{w}})$ .

5940 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch

5941 error listed above is returned.

5942 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with

5943 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5944 If an empty `GrB_Scalar`  $\tilde{\mathbf{s}}$  is provided ( $\mathbf{nvals}(\tilde{\mathbf{s}}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.

5945 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

5946 `val` with the same domain as  $\tilde{\mathbf{s}}$  and set `val = val( $\tilde{\mathbf{s}}$ )`.

5947 We are now ready to carry out the apply and any additional associated operations. We describe

5948 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.

#### 5982 4.3.8.4 apply: Matrix-BinaryOp variants[Scott: NEW CONTENT]

5983 Computes the transformation of the values of the stored elements of a matrix using a binary  
5984 operator and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the  
5985 first argument to the binary operator and stored elements of the matrix are passed as the second  
5986 argument. In the *bind-second* variant, the elements of the matrix are passed as the first argument  
5987 and the specified scalar value is passed as the second argument. The scalar can be passed either as  
5988 a non-opaque variable or as a GrB\_Scalar object.

#### 5989 C Syntax

```
5990 // bind-first + scalar value
5991 GrB_Info GrB_apply(GrB_Matrix      C,
5992                   const GrB_Matrix Mask,
5993                   const GrB_BinaryOp accum,
5994                   const GrB_BinaryOp op,
5995                   <type>           val,
5996                   const GrB_Matrix A,
5997                   const GrB_Descriptor desc);
```

```
5998 // bind-first + GraphBLAS scalar
5999 GrB_Info GrB_apply(GrB_Matrix      C,
6000                   const GrB_Matrix Mask,
6001                   const GrB_BinaryOp accum,
6002                   const GrB_BinaryOp op,
6003                   const GrB_Scalar s,
6004                   const GrB_Matrix A,
6005                   const GrB_Descriptor desc);
```

```
6006 // bind-second + scalar value
6007 GrB_Info GrB_apply(GrB_Matrix      C,
6008                   const GrB_Matrix Mask,
6009                   const GrB_BinaryOp accum,
6010                   const GrB_BinaryOp op,
6011                   const GrB_Matrix A,
6012                   <type>           val,
6013                   const GrB_Descriptor desc);
```

```
6014 // bind-second + GraphBLAS scalar
6015 GrB_Info GrB_apply(GrB_Matrix      C,
6016                   const GrB_Matrix Mask,
6017                   const GrB_BinaryOp accum,
6018                   const GrB_BinaryOp op,
6019                   const GrB_Matrix A,
```

```

6020         const GrB_Scalar      s,
6021         const GrB_Descriptor desc);

```

## 6022 Parameters

6023     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
6024     that may be accumulated with the result of the apply operation. On output, the  
6025     matrix holds the results of the operation.

6026     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
6027     stored into the output matrix C. The mask dimensions must match those of the  
6028     matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
6029     of the Mask matrix must be of type `bool` or any of the predefined “built-in” types  
6030     in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
6031     dimensions of C), `GrB_NULL` should be specified.

6032     **accum** (IN) An optional binary operator used for accumulating entries into existing C  
6033     entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
6034     specified.

6035     **op** (IN) A binary operator applied to each element of input matrix, A, with the element  
6036     of the input matrix used as the left-hand argument, and the scalar value, `val`, used  
6037     as the right-hand argument.

6038     **A** (IN) The GraphBLAS matrix whose elements are passed to the binary operator as  
6039     the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)  
6040     argument in the *bind-second* variant.

6041     **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)  
6042     argument in the *bind-first* variant, or the right-hand (second) argument in the  
6043     *bind-second* variant.

6044     **s** (IN) GraphBLAS scalar value that is passed to the binary operator as the left-hand  
6045     (first) argument in the *bind-first* variant, or the right-hand (second) argument in  
6046     the *bind-second* variant. It must not be empty.

6047     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
6048     should be specified. Non-default field/value pairs are listed as follows:

6049

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.

## 6077 Description

6078 This variant of `GrB_apply` computes the result of applying a binary operator to the elements of a  
6079 GraphBLAS matrix each composed with a scalar constant, `val` or `s`:

6080                   bind-first:      $C = f(\text{val}, A)$  or  $C = f(s, A)$

6081                   bind-second:    $C = f(A, \text{val})$  or  $C = f(A, s)$ ,

6082 or if an optional binary accumulation operator ( $\odot$ ) is provided:

6083                   bind-first:      $C = C \odot f(\text{val}, A)$  or  $C = C \odot f(s, A)$

6084                   bind-second:    $C = C \odot f(A, \text{val})$  or  $C = C \odot f(A, s)$ .

6085 Logically, this operation occurs in three steps:

6086       **Setup** The internal matrices and mask used in the computation are formed and their domains  
6087               and dimensions are tested for compatibility.

6088       **Compute** The indicated computations are carried out.

6089       **Output** The result is written into the output matrix, possibly under control of a mask.

6090 Up to three argument matrices are used in the `GrB_apply` operation:

- 6091     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6092     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)
- 6093     3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6094 The argument scalar, matrices, binary operator and the accumulation operator (if provided) are  
6095 tested for domain compatibility as follows:

- 6096     1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
6097       must be from one of the pre-defined types of Table 3.2.
- 6098     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the binary operator.
- 6099     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
6100       of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the binary operator must be compatible with  
6101        $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 6102     4.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.
- 6103     5. If bind-first:  
6104       (a)  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the binary operator.

6105 (b) If the non-opaque scalar  $\text{val}$  is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$   
 6106 of the binary operator.

6107 (c) If the `GrB_Scalar`  $s$  is provided, then  $\mathbf{D}(s)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the  
 6108 binary operator.

6109 6. If `bind-second`:

6110 (a)  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.

6111 (b) If the non-opaque scalar  $\text{val}$  is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$   
 6112 of the binary operator.

6113 (c) If the `GrB_Scalar`  $s$  is provided, then  $\mathbf{D}(s)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the  
 6114 binary operator.

6115 Two domains are compatible with each other if values from one domain can be cast to values in  
 6116 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 6117 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 6118 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
 6119 error listed above is returned.

6120 From the argument matrices, the internal matrices, mask, and index arrays used in the computation  
 6121 are formed ( $\leftarrow$  denotes copy):

6122 1. Matrix  $\tilde{C} \leftarrow C$ .

6123 2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument `Mask` as follows:

6124 (a) If `Mask` = `GrB_NULL`, then  $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$   
 6125  $j < \mathbf{ncols}(C)\} \rangle$ .

6126 (b) If `Mask`  $\neq$  `GrB_NULL`,

6127 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
 6128  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,

6129 ii. Otherwise,  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
 6130  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .

6131 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{M} \leftarrow \neg \tilde{M}$ .

6132 3. Matrix  $\tilde{A}$  is computed from argument `A` as follows:

6133 `bind-first`:  $\tilde{A} \leftarrow \text{desc}[\text{GrB\_INP1}].\text{GrB\_TRAN} ? A^T : A$

6134 `bind-second`:  $\tilde{A} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? A^T : A$

6135 4. Scalar  $\tilde{s} \leftarrow s$  (`GraphBLAS` scalar case).

6136 The internal matrices and mask are checked for dimension compatibility. The following conditions  
 6137 must hold:

6138 1.  $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{M})$ .

6139 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .

6140 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .

6141 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

6142 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6143 error listed above is returned.

6144 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6145 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6146 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6147 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable  
6148  $\mathbf{val}$  with the same domain as  $\tilde{s}$  and set  $\mathbf{val} = \mathbf{val}(\tilde{s})$ .

6149 We are now ready to carry out the apply and any additional associated operations. We describe  
6150 this in terms of two intermediate matrices:

- 6151 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the binary operator to the input matrix  $\tilde{\mathbf{A}}$ .
- 6152 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6153 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as one of the following:

6154 bind-first:  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\mathbf{val}, \tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

6155 bind-second:  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j), \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

6156 where  $f = \mathbf{f}(\mathbf{op})$ .

6157 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6158 • If  $\mathbf{accum} = \mathbf{GrB\_NULL}$ , then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6159 • If  $\mathbf{accum}$  is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6160 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6161 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6162 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$6163 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$6164 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$6165 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

6166 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

6169 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 6170 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 6171 mask which acts as a “write mask”.

- 6172 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
 6173 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6174 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6175 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 6176 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 6177 mask are unchanged:

$$6178 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6179 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 6180 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 6181 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
 6182 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 6183 sequence.

#### 6184 4.3.8.5 apply: Vector index unary operator variant[Scott: NEW CONTENT]

6185 Computes the transformation of the values of the stored elements of a vector using an index unary  
 6186 operator that is a function of the stored value, its location indices, and an user provided scalar  
 6187 value. The scalar can be passed either as a non-opaque variable or as a GrB\_Scalar object.

#### 6188 C Syntax

```
6189      GrB_Info GrB_apply(GrB_Vector          w,
6190                        const GrB_Vector      mask,
6191                        const GrB_BinaryOp    accum,
6192                        const GrB_IndexUnaryOp op,
6193                        const GrB_Vector      u,
6194                        <type>                val,
6195                        const GrB_Descriptor  desc);
```

```
6196      GrB_Info GrB_apply(GrB_Vector          w,
6197                        const GrB_Vector      mask,
6198                        const GrB_BinaryOp    accum,
6199                        const GrB_IndexUnaryOp op,
6200                        const GrB_Vector      u,
6201                        const GrB_Scalar      s,
6202                        const GrB_Descriptor  desc);
```



## Parameters

**w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values that may be accumulated with the result of the apply operation. On output, this vector holds the results of the operation.

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

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

**op** (IN) An index unary operator,  $F_i = \langle D_{out}, D_{in_1}, \mathbf{D}(\mathbf{GrB\_Index}), D_{in_2}, f_i \rangle$ , applied to each element stored in the input vector, **u**. It is a function of the stored element’s value, its location index, and a user supplied scalar value (either **s** or **val**).

**u** (IN) The GraphBLAS vector whose elements are passed to the index unary operator.

**val** (IN) An additional scalar value that is passed to the index unary operator.

**s** (IN) An additional GraphBLAS scalar that is passed to the index unary operator. It must not be empty.

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

Param	Field	Value	Description
<b>w</b>	<b>GrB_OUTP</b>	<b>GrB_REPLACE</b>	Output vector <b>w</b> is cleared (all elements removed) before the result is stored in it.
<b>mask</b>	<b>GrB_MASK</b>	<b>GrB_STRUCTURE</b>	The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined.
<b>mask</b>	<b>GrB_MASK</b>	<b>GrB_COMP</b>	Use the complement of <b>mask</b> .

## Return Values

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

6234                   GrB\_PANIC   Unknown internal error.

6235           GrB\_INVALID\_OBJECT   This is returned in any execution mode whenever one of the  
6236                                   opaque GraphBLAS objects (input or output) is in an invalid  
6237                                   state caused by a previous execution error. Call `GrB_error()` to  
6238                                   access any error messages generated by the implementation.

6239           GrB\_OUT\_OF\_MEMORY   Not enough memory available for operation.

6240   GrB\_UNINITIALIZED\_OBJECT   One or more of the GraphBLAS objects has not been initialized  
6241                                   by a call to `new` (or another constructor).

6242   GrB\_DIMENSION\_MISMATCH   `mask`, `w` and/or `u` dimensions are incompatible.

6243   GrB\_DOMAIN\_MISMATCH   The domains of the various vectors are incompatible with the cor-  
6244                                   responding domains of the accumulation operator or index unary  
6245                                   operator, or the mask's domain is not compatible with `bool` (in  
6246                                   the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6247           GrB\_EMPTY\_OBJECT   The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and  
6248                                   therefore a value cannot be passed to the index unary operator.

## 6249   Description

6250   This variant of `GrB_apply` computes the result of applying an index unary operator to the elements  
6251   of a GraphBLAS vector each composed with the element's index and a scalar constant, `val` or `s`:

$$6252 \qquad w = f_i(u, \mathbf{ind}(u), 0, \text{val}) \text{ or } w = f_i(u, \mathbf{ind}(u), 0, s),$$

6253   or if an optional binary accumulation operator ( $\odot$ ) is provided:

$$6254 \qquad w = w \odot f_i(u, \mathbf{ind}(u), 0, \text{val}) \text{ or } w = w \odot f_i(u, \mathbf{ind}(u), 0, s).$$

6255   Logically, this operation occurs in three steps:

6256       **Setup**   The internal vectors and mask used in the computation are formed and their domains  
6257                   and dimensions are tested for compatibility.

6258       **Compute**   The indicated computations are carried out.

6259       **Output**   The result is written into the output vector, possibly under control of a mask.

6260   Up to three argument vectors are used in this `GrB_apply` operation:

- 6261   1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6262   2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)

6263 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6264 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)  
6265 are tested for domain compatibility as follows:

- 6266 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
6267 must be from one of the pre-defined types of Table 3.2.
- 6268 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the index unary operator.
- 6269 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
6270 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the index unary operator must be compatible  
6271 with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 6272 4.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the index unary operator.
- 6273 5. If the non-opaque scalar `val` is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of  
6274 the index unary operator.
- 6275 6. If the `GrB_Scalar` `s` is provided, then  $\mathbf{D}(\mathbf{s})$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the index  
6276 unary operator.

6277 Two domains are compatible with each other if values from one domain can be cast to values in  
6278 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6279 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6280 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
6281 error listed above is returned.

6282 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
6283 denotes copy):

- 6284 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 6285 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 6286 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 6287 (b) If `mask`  $\neq$  `GrB_NULL`,
    - 6288 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,
    - 6289 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 6290 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 6291 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 6292 4. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).

6293 The internal vectors and masks are checked for dimension compatibility. The following conditions  
6294 must hold:

6295 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

6296 2.  $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$ .

6297 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6298 error listed above is returned.

6299 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6300 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6301 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6302 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided ( $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable `val`  
6303 with the same domain as  $\tilde{s}$  and set `val = val( $\tilde{s}$ )`.

6304 We are now ready to carry out the apply and any additional associated operations. We describe  
6305 this in terms of two intermediate vectors:

- 6306 •  $\tilde{\mathbf{t}}$ : The vector holding the result from applying the index unary operator to the input vector  
6307  $\tilde{\mathbf{u}}$ .
- 6308 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6309 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$6310 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f_i(\tilde{\mathbf{u}}(i), [i], 0, \text{val})) \mid i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,$$

6311 where  $f_i = \mathbf{f}(\text{op})$ .

6312 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6313 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 6314 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$6315 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6316 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
6317 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 6318 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 6319 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6320 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6321 \quad & \\ 6322 \end{aligned}$$

6323 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

6324 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
6325 using what is called a *standard vector mask and replace*. This is carried out under control of the  
6326 mask which acts as a “write mask”.

- If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $w$  on input to this operation are deleted and the content of the new output vector,  $w$ , is defined as,

$$L(w) = \{(i, z_i) : i \in (\text{ind}(\tilde{z}) \cap \text{ind}(\tilde{m}))\}.$$

- If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{z}$  indicated by the mask are copied into the result vector,  $w$ , and elements of  $w$  that fall outside the set indicated by the mask are unchanged:

$$L(w) = \{(i, w_i) : i \in (\text{ind}(w) \cap \text{ind}(\neg\tilde{m}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{z}) \cap \text{ind}(\tilde{m}))\}.$$

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

#### 4.3.8.6 apply: Matrix index unary operator variant[Scott: NEW CONTENT]

Computes the transformation of the values of the stored elements of a matrix using an index unary operator that is a function of the stored value, its location indices, and an user provided scalar value. The scalar can be passed either as a non-opaque variable or as a GrB\_Scalar object.

#### C Syntax

```
GrB_Info GrB_apply(GrB_Matrix      C,
                  const GrB_Matrix  Mask,
                  const GrB_BinaryOp accum,
                  const GrB_IndexUnaryOp op,
                  const GrB_Matrix  A,
                  <type>            val,
                  const GrB_Descriptor desc);
```

```
GrB_Info GrB_apply(GrB_Matrix      C,
                  const GrB_Matrix  Mask,
                  const GrB_BinaryOp accum,
                  const GrB_IndexUnaryOp op,
                  const GrB_Matrix  A,
                  const GrB_Scalar  s,
                  const GrB_Descriptor desc);
```

#### Parameters

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

6362       Mask (IN) An optional “write” mask that controls which results from this operation are  
6363       stored into the output matrix C. The mask dimensions must match those of the  
6364       matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
6365       of the Mask matrix must be of type **bool** or any of the predefined “built-in” types  
6366       in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6367       dimensions of C), GrB\_NULL should be specified.

6368       accum (IN) An optional binary operator used for accumulating entries into existing C  
6369       entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
6370       specified.

6371       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  
6372       to each element stored in the input matrix, A. It is a function of the stored element’s  
6373       value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6374       A (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-  
6375       ator.

6376       val (IN) An additional scalar value that is passed to the index unary operator.

6377       s (IN) An additional GraphBLAS scalar that is passed to the index unary operator.

6378       desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
6379       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.

## 6382 Return Values

6383       GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
6384       blocking mode, this indicates that the compatibility tests on di-  
6385       mensions and domains for the input arguments passed successfully.  
6386       Either way, output matrix C is ready to be used in the next method  
6387       of the sequence.

6388       GrB\_PANIC Unknown internal error.

6389       GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
6390       GraphBLAS objects (input or output) is in an invalid state caused

6391 by a previous execution error. Call `GrB_error()` to access any error  
 6392 messages generated by the implementation.

6393 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

6394 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 6395 a call to `new` (or another constructor).

6396 **GrB\_DIMENSION\_MISMATCH** `mask`, `w` and/or `u` dimensions are incompatible.

6397 **GrB\_DOMAIN\_MISMATCH** The domains of the various matrices are incompatible with the  
 6398 corresponding domains of the accumulation operator or index unary  
 6399 operator, or the mask's domain is not compatible with `bool` (in the  
 6400 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6401 **GrB\_EMPTY\_OBJECT** The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and  
 6402 therefore a value cannot be passed to the index unary operator.

## 6403 Description

6404 This variant of `GrB_apply` computes the result of applying a index unary operator to the elements  
 6405 of a GraphBLAS matrix each composed with the elements row and column indices, and a scalar  
 6406 constant, `val` or `s`:

$$6407 \quad C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s),$$

6408 or if an optional binary accumulation operator ( $\odot$ ) is provided:

$$6409 \quad C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s).$$

6410 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional  
 6411 indices, respectively.

6412 Logically, this operation occurs in three steps:

6413 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 6414 and dimensions are tested for compatibility.

6415 **Compute** The indicated computations are carried out.

6416 **Output** The result is written into the output matrix, possibly under control of a mask.

6417 Up to three argument matrices are used in the `GrB_apply` operation:

- 6418 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6419 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)

6420 3.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

6421 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)  
6422 are tested for domain compatibility as follows:

- 6423 1. If **Mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\mathbf{Mask})$   
6424 must be from one of the pre-defined types of Table 3.2.
- 6425 2.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator.
- 6426 3. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
6427 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator must be compatible  
6428 with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accumulation operator.
- 6429 4.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the index unary operator.
- 6430 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  
6431 the index unary operator.
- 6432 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  
6433 unary operator.

6434 Two domains are compatible with each other if values from one domain can be cast to values in  
6435 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6436 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6437 any compatibility rule above is violated, execution of **GrB\_apply** ends and the domain mismatch  
6438 error listed above is returned.

6439 From the argument matrices, the internal matrices, **mask**, and index arrays used in the computation  
6440 are formed ( $\leftarrow$  denotes copy):

- 6441 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 6442 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument **Mask** as follows:
  - 6443 (a) If **Mask** = **GrB\_NULL**, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
6444  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 6445 (b) If **Mask**  $\neq$  **GrB\_NULL**,
    - 6446 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
6447  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 6448 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
6449  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 6450 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 6451 3. Matrix  $\tilde{\mathbf{A}}$  is computed from argument **A** as follows:
 
$$6452 \quad \tilde{\mathbf{A}} \leftarrow \mathbf{desc[GrB_INP0].GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$$
- 6453 4. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).



6454 The internal matrices and mask are checked for dimension compatibility. The following conditions  
6455 must hold:

- 6456 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 6457 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 6458 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
- 6459 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

6460 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6461 error listed above is returned.

6462 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6463 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6464 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6465 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable  
6466  $\mathbf{val}$  with the same domain as  $\tilde{s}$  and set  $\mathbf{val} = \mathbf{val}(\tilde{s})$ .

6467 We are now ready to carry out the apply and any additional associated operations. We describe  
6468 this in terms of two intermediate matrices:

- 6469 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the index unary operator to the input matrix  
6470  $\tilde{\mathbf{A}}$ .
- 6471 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6472 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$6473 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f_i(\tilde{\mathbf{A}}(i, j), i, j, \mathbf{val})) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

6474 where  $f_i = \mathbf{f}(\mathbf{op})$ .

6475 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6476 • If  $\mathbf{accum} = \mathbf{GrB\_NULL}$ , then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6477 • If  $\mathbf{accum}$  is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6478 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6479 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6480 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 6481 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6482 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6483 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6484 \end{aligned}$$

6486 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

6487 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
6488 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
6489 mask which acts as a “write mask”.

- 6490 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
6491 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6492 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6493 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
6494 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
6495 mask are unchanged:

$$6496 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6497 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
6498 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
6499 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
6500 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
6501 sequence.

#### 6502 4.3.9 select:

6503 Apply a select operator to the stored elements of an object to determine whether or not to keep  
6504 them.

##### 6505 4.3.9.1 select: Vector variant[Scott: NEW CONTENT]

6506 Apply a select operator (an index unary operator) to the elements of a vector.

#### 6507 C Syntax

```
6508 // scalar value variant
6509 GrB_Info GrB_select(GrB_Vector          w,
6510                    const GrB_Vector     mask,
6511                    const GrB_BinaryOp   accum,
6512                    const GrB_IndexUnaryOp op,
6513                    const GrB_Vector     u,
6514                    <type>               val,
6515                    const GrB_Descriptor desc);
6516
6517 // GraphBLAS scalar variant
6518 GrB_Info GrB_select(GrB_Vector          w,
6519                    const GrB_Vector     mask,
```

```

6520         const GrB_BinaryOp      accum,
6521         const GrB_IndexUnaryOp  op,
6522         const GrB_Vector        u,
6523         const GrB_Scalar        s,
6524         const GrB_Descriptor    desc);
6525

```

## Parameters

**w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values that may be accumulated with the result of the select 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 GraphBLAS scalar that is passed to the index unary operator. It must not be empty.

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

Param	Field	Value	Description
<b>w</b>	<b>GrB_OUTP</b>	<b>GrB_REPLACE</b>	Output vector <b>w</b> is cleared (all elements removed) before the result is stored in it.
<b>mask</b>	<b>GrB_MASK</b>	<b>GrB_STRUCTURE</b>	The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined.
<b>mask</b>	<b>GrB_MASK</b>	<b>GrB_COMP</b>	Use the complement of <b>mask</b> .

## 6551 Return Values

6552	<b>GrB_SUCCESS</b>	In blocking mode, the operation completed successfully. In non-
6553		blocking mode, this indicates that the compatibility tests on di-
6554		mensions and domains for the input arguments passed success-
6555		fully. Either way, output vector <b>w</b> is ready to be used in the next
6556		method of the sequence.
6557	<b>GrB_PANIC</b>	Unknown internal error.
6558	<b>GrB_INVALID_OBJECT</b>	This is returned in any execution mode whenever one of the
6559		opaque GraphBLAS objects (input or output) is in an invalid
6560		state caused by a previous execution error. Call <b>GrB_error()</b> to
6561		access any error messages generated by the implementation.
6562	<b>GrB_OUT_OF_MEMORY</b>	Not enough memory available for operation.
6563	<b>GrB_UNINITIALIZED_OBJECT</b>	One or more of the GraphBLAS objects has not been initialized
6564		by a call to one of its constructors.
6565	<b>GrB_DIMENSION_MISMATCH</b>	<b>mask</b> , <b>w</b> and/or <b>u</b> dimensions are incompatible.
6566	<b>GrB_DOMAIN_MISMATCH</b>	The domains of the various vectors are incompatible with the cor-
6567		responding domains of the accumulation operator or index unary
6568		operator, or the mask's domain is not compatible with <b>bool</b> (in
6569		the case where <b>desc[GrB_MASK].GrB_STRUCTURE</b> is not set).
6570	<b>GrB_EMPTY_OBJECT</b>	The <b>GrB_Scalar</b> <b>s</b> used in the call is empty ( <b>nvals(s) = 0</b> ) and
6571		therefore a value cannot be passed to the index unary operator.

## 6572 Description

6573 This variant of **GrB\_select** computes the result of applying a index unary operator to select the  
6574 elements of the input GraphBLAS vector. The operator takes, as input, the value of each stored  
6575 element, along with the element's index and a scalar constant – either **val** or **s**. The corresponding  
6576 element of the input vector is selected (kept) if the function evaluates to **true** when cast to **bool**.  
6577 This acts like a functional mask on the input vector as follows:

$$6578 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle,$$

$$6579 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle.$$

6580 Correspondingly, if a **GrB\_Scalar**, **s**, is provided:

$$6581 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle,$$

$$6582 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle.$$

6583 Logically, this operation occurs in three steps:

6584     **Setup** The internal vectors and mask used in the computation are formed and their domains  
6585             and dimensions are tested for compatibility.

6586     **Compute** The indicated computations are carried out.

6587     **Output** The result is written into the output vector, possibly under control of a mask.

6588 Up to three argument vectors are used in this GrB\_select operation:

- 6589     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$   
6590     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)  
6591     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6592 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)  
6593 are tested for domain compatibility as follows:

- 6594     1. If **mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\mathbf{mask})$   
6595         must be from one of the pre-defined types of Table 3.2.  
6596     2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .  
6597     3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
6598         of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accu-  
6599         mulation operator.  
6600     4.  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator must be from one of the pre-defined types of Table 3.2;  
6601         i.e., castable to **bool**.  
6602     5.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the index unary operator.  
6603     6.  $\mathbf{D}(\mathbf{val})$  or  $\mathbf{D}(\mathbf{s})$ , depending on the signature of the method, must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$   
6604         of the index unary operator.

6605 Two domains are compatible with each other if values from one domain can be cast to values in  
6606 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6607 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6608 any compatibility rule above is violated, execution of GrB\_select ends and the domain mismatch  
6609 error listed above is returned.

6610 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
6611 denotes copy):

- 6612     1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .  
6613     2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument **mask** as follows:

6614 (a) If  $\text{mask} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .  
6615 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,  
6616 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,  
6617 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .  
6618 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .  
6619 3. Vector  $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$ .  
6620 4. Scalar  $\widetilde{s} \leftarrow s$  (GrB\_Scalar version only).

6621 The internal vectors and masks are checked for dimension compatibility. The following conditions  
6622 must hold:

- 6623 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 6624 2.  $\text{size}(\widetilde{\mathbf{u}}) = \text{size}(\widetilde{\mathbf{w}})$ .

6625 If any compatibility rule above is violated, execution of `GrB_select` ends and the dimension mismatch  
6626 error listed above is returned.

6627 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6628 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6629 If an empty `GrB_Scalar`  $\widetilde{s}$  is provided (i.e.,  $\text{nvals}(\widetilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6630 If a non-empty `GrB_Scalar`,  $\widetilde{s}$ , is provided (i.e.,  $\text{nvals}(\widetilde{s}) = 1$ ), we then create an internal variable  
6631 `val` with the same domain as  $\widetilde{s}$  and set  $\text{val} = \text{val}(\widetilde{s})$ .

6632 We are now ready to carry out the `select` and any additional associated operations. We describe  
6633 this in terms of two intermediate vectors:

- 6634 •  $\widetilde{\mathbf{t}}$ : The vector holding the result from applying the index unary operator to the input vector  
6635  $\widetilde{\mathbf{u}}$ .
- 6636 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6637 The intermediate vector,  $\widetilde{\mathbf{t}}$ , is created as follows:

$$6638 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{u}}), \{(i, \widetilde{\mathbf{u}}(i), : i \in \text{ind}(\widetilde{\mathbf{u}}) \wedge (\text{bool})f_i(\widetilde{\mathbf{u}}(i), i, 0, \text{val}) = \text{true})\} \rangle,$$

6639 where  $f_i = \mathbf{f}(\text{op})$ .

6640 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6641 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$ .
- 6642 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{z}}$  is defined as

$$6643 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

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);
```

```

6679 // GraphBLAS scalar variant
6680 GrB_Info GrB_select(GrB_Matrix          C,
6681                    const GrB_Matrix     Mask,
6682                    const GrB_BinaryOp    accum,
6683                    const GrB_IndexUnaryOp op,
6684                    const GrB_Matrix      A,
6685                    const GrB_Scalar      s,
6686                    const GrB_Descriptor   desc);

```

## 6687 Parameters

6688     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
6689     that may be accumulated with the result of the select operation. On output, the  
6690     matrix holds the results of the operation.

6691     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
6692     stored into the output matrix **C**. The mask dimensions must match those of the  
6693     matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6694     of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
6695     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6696     dimensions of **C**), **GrB\_NULL** should be specified.

6697     **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
6698     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6699     specified.

6700     **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  
6701     to each element stored in the input matrix, **A**. It is a function of the stored element’s  
6702     value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6703     **A** (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-  
6704     ator.

6705     **val** (IN) An additional scalar value that is passed to the index unary operator.

6706     **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must  
6707     not be empty.

6708     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
6709     should be specified. Non-default field/value pairs are listed as follows:

6710



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:

6740  $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$ , or  
 6741  $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$ .

6742 Correspondingly, if a GrB\_Scalar,  $s$ , is provided:

6743  $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$ , or  
 6744  $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$ .

6745 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional  
 6746 indices, respectively.

6747 Logically, this operation occurs in three steps:

6748 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 6749 and dimensions are tested for compatibility.

6750 **Compute** The indicated computations are carried out.

6751 **Output** The result is written into the output matrix, possibly under control of a mask.

6752 Up to three argument matrices are used in the GrB\_select operation:

- 6753 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6754 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)
- 6755 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6756 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)  
 6757 are tested for domain compatibility as follows:

- 6758 1. If **Mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{Mask})$   
 6759 must be from one of the pre-defined types of Table 3.2.
- 6760 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$ .
- 6761 3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 6762 of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 6763 mulation operator.
- 6764 4.  $\mathbf{D}_{out}(\text{op})$  of the index unary operator must be from one of the pre-defined types of Table 3.2;  
 6765 i.e., castable to **bool**.
- 6766 5.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the index unary operator.
- 6767 6.  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(s)$ , depending on the signature of the method, must be compatible with  $\mathbf{D}_{in_2}(\text{op})$   
 6768 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:

- 6803 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the index unary operator to the input matrix  
6804  $\tilde{\mathbf{A}}$ .
- 6805 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6806 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$6807 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \\ \{(i, j, \tilde{\mathbf{A}}(i, j) : i, j \in \mathbf{ind}(\tilde{\mathbf{A}}) \wedge (\text{bool})f_i(\tilde{\mathbf{A}}(i, j), i, j, \text{val}) = \text{true})\},$$

6808 where  $f_i = \mathbf{f}(\text{op})$ .

6809 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6810 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6811 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6812 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\}\rangle.$$

6813 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6814 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$6815 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6816 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6817 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6818 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6819$$

6820 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

6821 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
6822 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
6823 mask which acts as a “write mask”.

- 6824 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
6825 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6826 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6827 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
6828 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
6829 mask are unchanged:

$$6830 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6831 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
6832 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
6833 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
6834 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
6835 sequence.

### 6836 4.3.10 reduce: Perform a reduction across the elements of an object

6837 Computes the reduction of the values of the elements of a vector or matrix.

#### 6838 4.3.10.1 reduce: Standard matrix to vector variant

6839 This performs a reduction across rows of a matrix to produce a vector. If reduction down columns  
6840 is desired, the input matrix should be transposed using the descriptor.

### 6841 C Syntax

```
6842     GrB_Info GrB_reduce(GrB_Vector      w,  
6843                        const GrB_Vector mask,  
6844                        const GrB_BinaryOp accum,  
6845                        const GrB_Monoid op,  
6846                        const GrB_Matrix A,  
6847                        const GrB_Descriptor desc);  
6848  
6849     GrB_Info GrB_reduce(GrB_Vector      w,  
6850                        const GrB_Vector mask,  
6851                        const GrB_BinaryOp accum,  
6852                        const GrB_BinaryOp op,  
6853                        const GrB_Matrix A,  
6854                        const GrB_Descriptor desc);
```

### 6855 Parameters

6856 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
6857 that may be accumulated with the result of the reduction operation. On output,  
6858 this vector holds the results of the operation.

6859 **mask** (IN) An optional “write” mask that controls which results from this operation are  
6860 stored into the output vector **w**. The mask dimensions must match those of the  
6861 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6862 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
6863 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6864 dimensions of **w**), **GrB\_NULL** should be specified.

6865 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
6866 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6867 specified.

6868 **op** (IN) The monoid or binary operator used in the element-wise reduction operation.  
6869 Depending on which type is passed, the following defines the binary operator with  
6870 one domain,  $F_b = \langle D, D, D, \oplus \rangle$ , that is used:

6871 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .  
6872 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  
6873 monoid is ignored.

6874 If  $\text{op}$  is a GrB\_BinaryOp, then all its domains must be the same. Furthermore, in  
6875 both cases  $\odot(\text{op})$  must be commutative and associative. Otherwise, the outcome  
6876 of the operation is undefined.

6877 A (IN) The GraphBLAS matrix on which reduction will be performed.

6878 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
6879 should be specified. Non-default field/value pairs are listed as follows:  
6880

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.

## 6882 Return Values

6883 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
6884 blocking mode, this indicates that the compatibility tests on di-  
6885 mensions and domains for the input arguments passed successfully.  
6886 Either way, output vector w is ready to be used in the next method  
6887 of the sequence.

6888 GrB\_PANIC Unknown internal error.

6889 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
6890 GraphBLAS objects (input or output) is in an invalid state caused  
6891 by a previous execution error. Call GrB\_error() to access any error  
6892 messages generated by the implementation.

6893 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

6894 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
6895 a call to new (or dup for vector parameters).

6896 GrB\_DIMENSION\_MISMATCH mask, w and/or u dimensions are incompatible.

6897 GrB\_DOMAIN\_MISMATCH Either the domains of the various vectors and matrices are incom-  
6898 patible with the corresponding domains of the accumulation oper-  
6899 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).

## 6903 Description

6904 This variant of `GrB_reduce` computes the result of performing a reduction across each of the rows  
 6905 of an input matrix:  $w(i) = \bigoplus A(i, :) \forall i$ ; or, if an optional binary accumulation operator is provided,  
 6906  $w(i) = w(i) \odot (\bigoplus A(i, :)) \forall i$ , where  $\bigoplus = \odot(F_b)$  and  $\odot = \odot(\text{accum})$ .

6907 Logically, this operation occurs in three steps:

6908     **Setup** The internal vector, matrix and mask used in the computation are formed and their  
 6909 domains and dimensions are tested for compatibility.

6910 **Compute** The indicated computations are carried out.

6911 **Output** The result is written into the output vector, possibly under control of a mask.

6912 Up to two vector and one matrix argument are used in this `GrB_reduce` operation:

- 6913 1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6914 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 6915 3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6916 The argument vector, matrix, reduction operator and accumulation operator (if provided) are tested  
 6917 for domain compatibility as follows:

- 6918 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
 6919 must be from one of the pre-defined types of Table 3.2.
- 6920 2.  $\mathbf{D}(w)$  must be compatible with the domain of the reduction binary operator,  $\mathbf{D}(F_b)$ .
- 6921 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 6922 of the accumulation operator and  $\mathbf{D}(F_b)$ , must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 6923 mulation operator.
- 6924 4.  $\mathbf{D}(A)$  must be compatible with the domain of the binary reduction operator,  $\mathbf{D}(F_b)$ .

6925 Two domains are compatible with each other if values from one domain can be cast to values in  
 6926 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 6927 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 6928 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch  
 6929 error listed above is returned.

6930 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
 6931 denotes copy):

- 6932 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 6933 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
- 6934 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
- 6935 (b) If `mask  $\neq$  GrB_NULL`,
- 6936 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
- 6937 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
- 6938 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 6939 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

6940 The internal vectors and masks are checked for dimension compatibility. The following conditions  
6941 must hold:

- 6942 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 6943 2.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .

6944 If any compatibility rule above is violated, execution of `GrB_reduce` ends and the dimension mis-  
6945 match error listed above is returned.

6946 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6947 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6948 We carry out the reduce and any additional associated operations. We describe this in terms of  
6949 two intermediate vectors:

- 6950 •  $\tilde{\mathbf{t}}$ : The vector holding the result from reducing along the rows of input matrix  $\tilde{\mathbf{A}}$ .
- 6951 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6952 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$6953 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, t_i) : \mathbf{ind}(\mathbf{A}(i, :)) \neq \emptyset\} \rangle.$$

6954 The value of each of its elements is computed by

$$6955 \quad t_i = \bigoplus_{j \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :))} \tilde{\mathbf{A}}(i, j),$$

6956 where  $\bigoplus = \odot(F_b)$ .

6957 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6958 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .



- If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ , using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a “write mask”.

- If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

- If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

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

#### 4.3.10.2 reduce: Vector-scalar variant[Scott: NEW CONTENT]

Reduce all stored values into a single scalar.

### C Syntax

```
// scalar value + monoid (only)
GrB_Info GrB_reduce(<type>          *val,
                    const GrB_BinaryOp accum,
                    const GrB_Monoid  op,
                    const GrB_Vector  u,
```

```

6992             const GrB_Descriptor desc);
6993
6994 // GraphBLAS Scalar + monoid
6995 GrB_Info GrB_reduce(GrB_Scalar      s,
6996                   const GrB_BinaryOp accum,
6997                   const GrB_Monoid   op,
6998                   const GrB_Vector   u,
6999                   const GrB_Descriptor desc);
7000
7001 // GraphBLAS Scalar + binary operator
7002 GrB_Info GrB_reduce(GrB_Scalar      s,
7003                   const GrB_BinaryOp accum,
7004                   const GrB_BinaryOp op,
7005                   const GrB_Vector   u,
7006                   const GrB_Descriptor desc);

```

## 7007 Parameters

7008 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides  
7009 a value that may be accumulated (optionally) with the result of the reduction  
7010 operation. On output, this scalar holds the results of the operation.

7011 **accum** (IN) An optional binary operator used for accumulating entries into an exist-  
7012 ing scalar (**s** or **val**) value. If assignment rather than accumulation is desired,  
7013 **GrB\_NULL** should be specified.

7014 **op** (IN) The monoid ( $M = \langle D, \oplus, 0 \rangle$ ) or binary operator ( $F_b = \langle D, D, D, \oplus \rangle$ ) used in  
7015 the reduction operation. The  $\oplus$  operator must be commutative and associative;  
7016 otherwise, the outcome of the operation is undefined.

7017 **u** (IN) The GraphBLAS vector on which reduction will be performed.

7018 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
7019 should be specified. Non-default field/value pairs are listed as follows:

7021	Param	Field	Value	Description
------	-------	-------	-------	-------------

7022 *Note:* This argument is defined for consistency with the other GraphBLAS opera-  
7023 tions. There are currently no non-default field/value pairs that can be set for this  
7024 operation.

## 7025 Return Values

7026 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
7027 cessfully, and the output scalar (**s** or **val**) is ready to be used in the  
7028 next method of the sequence.

7029                   GrB\_PANIC Unknown internal error.

7030           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
7031                   GraphBLAS objects (input or output) is in an invalid state caused  
7032                   by a previous execution error. Call GrB\_error() to access any error  
7033                   messages generated by the implementation.

7034           GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

7035 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
7036                   a call to a respective constructor.

7037           GrB\_NULL\_POINTER val pointer is NULL.

7038           GrB\_DOMAIN\_MISMATCH The domains of input and output arguments are incompatible with  
7039                   the corresponding domains of the accumulation operator, or reduce  
7040                   operator.

## 7041 Description

This variant of GrB\_reduce computes the result of performing a reduction across all of the stored elements of an input vector storing the result into either **s** or **val**. This corresponds to (shown here for the scalar value case only):

$$\text{val} = \begin{cases} \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i), & \text{or} \\ \text{val} \odot \left[ \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7042 where  $\bigoplus = \odot(\text{op})$  and  $\odot = \odot(\text{accum})$ .

7043 Logically, this operation occurs in three steps:

7044       **Setup** The internal vector used in the computation is formed and its domain is tested for  
7045                   compatibility.

7046       **Compute** The indicated computations are carried out.

7047       **Output** The result is written into the output scalar.

7048 One vector argument is used in this GrB\_reduce operation:

- 7049       1.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

7050 The output scalar, argument vector, reduction operator and accumulation operator (if provided)  
7051 are tested for domain compatibility as follows:

- 7052       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  
7053            $\mathbf{D}_{in_1}(\text{op})$  and  $\mathbf{D}_{in_2}(\text{op})$  from  $F_b$ ).

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  $\mathbf{D}_{out}(\text{accum})$  of the accumulation operator, and  $\mathbf{D}(\text{op})$  from  $M$  (or  $\mathbf{D}_{out}(\text{op})$  from  $F_b$ ) must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.

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  $F_b$ ).

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_reduce` ends and the domain mismatch error listed above is returned.

The number of values stored in the input, `u`, is checked. If there are no stored values in `u`, then one of the following occurs depending on the output variant:

$$\mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if } \text{accum} = \text{GrB\_NULL}, \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

or

$$\text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if } \text{accum} = \text{GrB\_NULL}, \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

where  $\mathbf{0}(\text{op})$  is the identity of the monoid. The operation returns immediately with `GrB_SUCCESS`.

For all other cases, the internal vector and scalar used in the computation is formed ( $\leftarrow$  denotes copy):

1. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

2. Scalar  $\tilde{s} \leftarrow \text{s}$  (GraphBLAS scalar case).

We are now ready to carry out the reduction and any additional associated operations. An intermediate scalar result  $t$  is computed as follows:

$$t = \bigoplus_{i \in \text{ind}(\tilde{\mathbf{u}})} \tilde{\mathbf{u}}(i),$$

where  $\oplus = \odot(\text{op})$ .

The final reduction value is computed as follows:

$$\mathbf{L}(\text{s}) \leftarrow \begin{cases} \{t\}, & \text{when } \text{accum} = \text{GrB\_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\text{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

or

$$\text{val} \leftarrow \begin{cases} t, & \text{when } \text{accum} = \text{GrB\_NULL, or} \\ \text{val} \odot t, & \text{otherwise;} \end{cases}$$

7081 In both GrB\_BLOCKING and GrB\_NONBLOCKING modes, the method exits with return value  
7082 GrB\_SUCCESS and the new contents of the output scalar is as defined above.

#### 7083 4.3.10.3 reduce: Matrix-scalar variant[Scott: NEW CONTENT]

7084 Reduce all stored values into a single scalar.

### 7085 C Syntax

```
7086 // scalar value + monoid (only)
7087 GrB_Info GrB_reduce(<type>          *val,
7088                   const GrB_BinaryOp accum,
7089                   const GrB_Monoid   op,
7090                   const GrB_Matrix   A,
7091                   const GrB_Descriptor desc);
7092
7093 // GraphBLAS Scalar + monoid
7094 GrB_Info GrB_reduce(GrB_Scalar      s,
7095                   const GrB_BinaryOp accum,
7096                   const GrB_Monoid   op,
7097                   const GrB_Matrix   A,
7098                   const GrB_Descriptor desc);
7099
7100 // GraphBLAS Scalar + binary operator
7101 GrB_Info GrB_reduce(GrB_Scalar      s,
7102                   const GrB_BinaryOp accum,
7103                   const GrB_BinaryOp op,
7104                   const GrB_Matrix   A,
7105                   const GrB_Descriptor desc);
```

### 7106 Parameters

7107 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides  
7108 a value that may be accumulated (optionally) with the result of the reduction  
7109 operation. On output, this scalar holds the results of the operation.

7110 **accum** (IN) An optional binary operator used for accumulating entries into existing (**s** or  
7111 **val**) value. If assignment rather than accumulation is desired, GrB\_NULL should  
7112 be specified.

7113 **op** (IN) The monoid ( $M = \langle D, \oplus, 0 \rangle$ ) or binary operator ( $F_b = \langle D, D, D, \oplus \rangle$ ) used in  
7114 the reduction operation. The  $\oplus$  operator must be commutative and associative;  
7115 otherwise, the outcome of the operation is undefined.

7116 **A** (IN) The GraphBLAS matrix on which the reduction will be performed.

7117 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
7118 should be specified. Non-default field/value pairs are listed as follows:  
7119

7120	Param	Field	Value	Description
------	-------	-------	-------	-------------

7121 *Note:* This argument is defined for consistency with the other GraphBLAS opera-  
7122 tions. There are currently no non-default field/value pairs that can be set for this  
7123 operation.

## 7124 Return Values

7125 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
7126 cessfully, and the output scalar (s or val) is ready to be used in the  
7127 next method of the sequence.

7128 GrB\_PANIC Unknown internal error.

7129 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
7130 GraphBLAS objects (input or output) is in an invalid state caused  
7131 by a previous execution error. Call GrB\_error() to access any error  
7132 messages generated by the implementation.

7133 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

7134 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
7135 a call to a respective constructor.

7136 GrB\_NULL\_POINTER val pointer is NULL.

7137 GrB\_DOMAIN\_MISMATCH The domains of input and output arguments are incompatible with  
7138 the corresponding domains of the accumulation operator, or reduce  
7139 operator.

## 7140 Description

This variant of GrB\_reduce computes the result of performing a reduction across all of the stored elements of an input matrix storing the result into either s or val. This corresponds to (shown here for the scalar value case only):

$$\text{val} = \begin{cases} \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j), & \text{or} \\ \text{val} \odot \left[ \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7141 where  $\bigoplus = \odot(\text{op})$  and  $\odot = \odot(\text{accum})$ .

7142 Logically, this operation occurs in three steps:

7143       **Setup** The internal matrix used in the computation is formed and its domain is tested for  
 7144       compatibility.

7145       **Compute** The indicated computations are carried out.

7146       **Output** The result is written into the output scalar.

7147   One matrix argument is used in this GrB\_reduce operation:

7148       1.  $A = \langle \mathbf{D}(A), \mathbf{size}(A), \mathbf{L}(A) = \{(i, j, A_{i,j})\} \rangle$

7149   The output scalar, argument matrix, reduction operator and accumulation operator (if provided)  
 7150   are tested for domain compatibility as follows:

7151       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  
 7152        $\mathbf{D}_{in_1}(\text{op})$  and  $\mathbf{D}_{in_2}(\text{op})$  from  $F_b$ ).

7153       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  
 7154        $\mathbf{D}_{out}(\text{accum})$  of the accumulation operator, and  $\mathbf{D}(\text{op})$  from  $M$  (or  $\mathbf{D}_{out}(\text{op})$  from  $F_b$ ) must  
 7155       be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.

7156       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  $F_b$ ).

7157   Two domains are compatible with each other if values from one domain can be cast to values in  
 7158   the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 7159   compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 7160   any compatibility rule above is violated, execution of GrB\_reduce ends and the domain mismatch  
 7161   error listed above is returned.

7162   The number of values stored in the input,  $A$ , is checked. If there are no stored values in  $A$ , then  
 7163   one of the following occurs depending on the output variant:

$$7164 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if accum = GrB\_NULL,} \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7165   or

$$7166 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if accum = GrB\_NULL,} \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7167   where  $\mathbf{0}(\text{op})$  is the identity of the monoid. The operation returns immediately with GrB\_SUCCESS.

7168   For all other cases, the internal matrix and scalar used in the computation is formed ( $\leftarrow$  denotes  
 7169   copy):

7170       1. Matrix  $\tilde{A} \leftarrow A$ .

7171       2. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).

7172 We are now ready to carry out the reduce and any additional associated operations. An intermediate  
 7173 scalar result  $t$  is computed as follows:

$$7174 \quad t = \bigoplus_{(i,j) \in \text{ind}(\tilde{\mathbf{A}})} \tilde{\mathbf{A}}(i,j),$$

7175 where  $\oplus = \odot(\text{op})$ .

7176 The final reduction value is computed as follows:

$$7177 \quad \mathbf{L}(\mathbf{s}) \leftarrow \begin{cases} \{t\}, & \text{when accum} = \text{GrB\_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\mathbf{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7178 or

$$7179 \quad \mathbf{val} \leftarrow \begin{cases} t, & \text{when accum} = \text{GrB\_NULL, or} \\ \mathbf{val} \odot t, & \text{otherwise;} \end{cases}$$

7180 In both GrB\_BLOCKING and GrB\_NONBLOCKING modes, the method exits with return value  
 7181 GrB\_SUCCESS and the new contents of the output scalar is as defined above.

#### 7182 4.3.11 transpose: Transpose rows and columns of a matrix

7183 This version computes a new matrix that is the transpose of the source matrix.

#### 7184 C Syntax

```
7185     GrB_Info GrB_transpose(GrB_Matrix      C,
7186                           const GrB_Matrix Mask,
7187                           const GrB_BinaryOp accum,
7188                           const GrB_Matrix A,
7189                           const GrB_Descriptor desc);
```

#### 7190 Parameters

7191 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
 7192 that may be accumulated with the result of the transpose operation. On output,  
 7193 the matrix holds the results of the operation.

7194 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
 7195 stored into the output matrix C. The mask dimensions must match those of the  
 7196 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
 7197 of the Mask matrix must be of type bool or any of the predefined “built-in” types  
 7198 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
 7199 dimensions of C), GrB\_NULL should be specified.



7200 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
7201 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
7202 specified.

7203 **A** (IN) The GraphBLAS matrix on which transposition will be performed.

7204 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
7205 should be specified. Non-default field/value pairs are listed as follows:  
7206

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.

## 7208 Return Values

7209 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
7210 blocking mode, this indicates that the compatibility tests on di-  
7211 mensions and domains for the input arguments passed successfully.  
7212 Either way, output matrix C is ready to be used in the next method  
7213 of the sequence.

7214 **GrB\_PANIC** Unknown internal error.

7215 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
7216 GraphBLAS objects (input or output) is in an invalid state caused  
7217 by a previous execution error. Call **GrB\_error()** to access any error  
7218 messages generated by the implementation.

7219 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

7220 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
7221 a call to **new** (or **Matrix\_dup** for matrix parameters).

7222 **GrB\_DIMENSION\_MISMATCH** mask, C and/or A dimensions are incompatible.

7223 **GrB\_DOMAIN\_MISMATCH** The domains of the various matrices are incompatible with the cor-  
7224 responding domains of the accumulation operator, or the mask's do-  
7225 main is not compatible with **bool** (in the case where **desc[GrB\_MASK].GrB\_STRUCTURE**  
7226 is not set).

## 7227 Description

7228 GrB\_transpose computes the result of performing a transpose of the input matrix:  $C = A^T$ ; or, if an  
 7229 optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot A^T$ . We note that the input matrix  
 7230 A can itself be optionally transposed before the operation, which would cause either an assignment  
 7231 from A to C or an accumulation of A into C.

7232 Logically, this operation occurs in three steps:

7233     **Setup** The internal matrix and mask used in the computation are formed and their domains  
 7234             and dimensions are tested for compatibility.

7235     **Compute** The indicated computations are carried out.

7236     **Output** The result is written into the output matrix, possibly under control of a mask.

7237 Up to three matrix arguments are used in this GrB\_transpose operation:

- 7238 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7239 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)
- 7240 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

7241 The argument matrices and accumulation operator (if provided) are tested for domain compatibility  
 7242 as follows:

- 7243 1. If Mask is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{Mask})$   
 7244     must be from one of the pre-defined types of Table 3.2.
- 7245 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$  of the input matrix.
- 7246 3. If accum is not GrB\_NULL, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 7247     of the accumulation operator and  $\mathbf{D}(A)$  of the input matrix must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
 7248     of the accumulation operator.

7249 Two domains are compatible with each other if values from one domain can be cast to values in  
 7250 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 7251 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 7252 any compatibility rule above is violated, execution of GrB\_transpose ends and the domain mismatch  
 7253 error listed above is returned.

7254 From the argument matrices, the internal matrices and mask used in the computation are formed  
 7255 ( $\leftarrow$  denotes copy):

- 7256 1. Matrix  $\tilde{C} \leftarrow C$ .
- 7257 2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument Mask as follows:

- 7258 (a) If  $\text{Mask} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
7259  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
- 7260 (b) If  $\text{Mask} \neq \text{GrB\_NULL}$ ,
- 7261 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
7262  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
- 7263 ii. Otherwise,  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
7264  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
- 7265 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{M}} \leftarrow \neg \widetilde{\mathbf{M}}$ .
- 7266 3. Matrix  $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

7267 The internal matrices and masks are checked for dimension compatibility. The following conditions  
7268 must hold:

- 7269 1.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$ .
- 7270 2.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$ .
- 7271 3.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{A}})$ .
- 7272 4.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$ .

7273 If any compatibility rule above is violated, execution of `GrB_transpose` ends and the dimension  
7274 mismatch error listed above is returned.

7275 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
7276 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

7277 We are now ready to carry out the matrix transposition and any additional associated operations.  
7278 We describe this in terms of two intermediate matrices:

- 7279 •  $\widetilde{\mathbf{T}}$ : The matrix holding the transpose of  $\widetilde{\mathbf{A}}$ .
- 7280 •  $\widetilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

7281 The intermediate matrix

$$7282 \quad \widetilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \{(j, i, A_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{A}})\} \rangle$$

7283 is created.

7284 The intermediate matrix  $\widetilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 7285 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$ .
- 7286 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{Z}}$  is defined as

$$7287 \quad \widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\} \rangle.$$

The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ , using what is called a *standard matrix mask and replace*. This is carried out under control of the mask which acts as a “write mask”.

- If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$\mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

#### 4.3.12 kronecker: Kronecker product of two matrices

Computes the Kronecker product of two matrices. The result is a matrix.

#### C Syntax

```
GrB_Info GrB_kronecker(GrB_Matrix      C,
                        const GrB_Matrix Mask,
                        const GrB_BinaryOp accum,
                        const GrB_Semiring op,
                        const GrB_Matrix A,
                        const GrB_Matrix B,
                        const GrB_Descriptor desc);
```

```

7322     GrB_Info GrB_kronecker(GrB_Matrix      C,
7323                           const GrB_Matrix  Mask,
7324                           const GrB_BinaryOp accum,
7325                           const GrB_Monoid   op,
7326                           const GrB_Matrix  A,
7327                           const GrB_Matrix  B,
7328                           const GrB_Descriptor desc);
7329
7330     GrB_Info GrB_kronecker(GrB_Matrix      C,
7331                           const GrB_Matrix  Mask,
7332                           const GrB_BinaryOp accum,
7333                           const GrB_BinaryOp op,
7334                           const GrB_Matrix  A,
7335                           const GrB_Matrix  B,
7336                           const GrB_Descriptor desc);

```

### 7337 Parameters

7338 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
7339 that may be accumulated with the result of the Kronecker product. On output,  
7340 the matrix holds the results of the operation.

7341 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
7342 stored into the output matrix C. The mask dimensions must match those of the  
7343 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
7344 of the Mask matrix must be of type bool or any of the predefined “built-in” types  
7345 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
7346 dimensions of C), GrB\_NULL should be specified.

7347 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
7348 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
7349 specified.

7350 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”  
7351 operation. Depending on which type is passed, the following defines the binary  
7352 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

7353 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

7354 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-  
7355 nored.

7356 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  
7357 is ignored.

7358 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
7359 product.

7360 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
7361 product.

7362 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
7363 should be specified. Non-default field/value pairs are listed as follows:  
7364

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.

## 7366 Return Values

7367 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
7368 blocking mode, this indicates that the compatibility tests on di-  
7369 mensions and domains for the input arguments passed successfully.  
7370 Either way, output matrix C is ready to be used in the next method  
7371 of the sequence.

7372 GrB\_PANIC Unknown internal error.

7373 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
7374 GraphBLAS objects (input or output) is in an invalid state caused  
7375 by a previous execution error. Call GrB\_error() to access any error  
7376 messages generated by the implementation.

7377 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

7378 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
7379 a call to new (or Matrix\_dup for matrix parameters).

7380 GrB\_DIMENSION\_MISMATCH Mask and/or matrix dimensions are incompatible.

7381 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
7382 corresponding domains of the binary operator (op) or accumulation  
7383 operator, or the mask's domain is not compatible with bool (in the  
7384 case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

## 7385 Description

7386 GrB\_kronecker computes the Kronecker product  $C = A \otimes B$  or, if an optional binary accumulation  
7387 operator ( $\odot$ ) is provided,  $C = C \odot (A \otimes B)$  (where matrices A and B can be optionally transposed).

7388 The Kronecker product is defined as follows:

7389

$$7390 \quad C = A \otimes B = \begin{bmatrix} A_{0,0} \otimes B & A_{0,1} \otimes B & \dots & A_{0,n_A-1} \otimes B \\ A_{1,0} \otimes B & A_{1,1} \otimes B & \dots & A_{1,n_A-1} \otimes B \\ \vdots & \vdots & \ddots & \vdots \\ A_{m_A-1,0} \otimes B & A_{m_A-1,1} \otimes B & \dots & A_{m_A-1,n_A-1} \otimes B \end{bmatrix}$$

7391 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  
7392 Kronecker product are defined as

$$7393 \quad C(i_A m_B + i_B, j_A n_B + j_B) = A_{i_A, j_A} \otimes B_{i_B, j_B},$$

7394 where  $\otimes$  is the multiplicative operator specified by the **op** parameter.

7395 Logically, this operation occurs in three steps:

7396 **Setup** The internal matrices and mask used in the computation are formed and their domains  
7397 and dimensions are tested for compatibility.

7398 **Compute** The indicated computations are carried out.

7399 **Output** The result is written into the output matrix, possibly under control of a mask.

7400 Up to four argument matrices are used in the **GrB\_kronecker** operation:

- 7401 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7402 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)
- 7403 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 7404 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

7405 The argument matrices, the "product" operator (**op**), and the accumulation operator (if provided)  
7406 are tested for domain compatibility as follows:

- 7407 1. If **Mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\text{Mask})$   
7408 must be from one of the pre-defined types of Table 3.2.
- 7409 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$ .
- 7410 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$ .
- 7411 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 7412 5. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
7413 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of **op** must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
7414 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.



7448 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}) \times \mathbf{nrows}(\tilde{\mathbf{B}}), \mathbf{ncols}(\tilde{\mathbf{A}}) \times \mathbf{ncols}(\tilde{\mathbf{B}}), \{(i, j, T_{ij}) \text{ where } i =$   
7449  $i_A \cdot m_B + i_B, j = j_A \cdot n_B + j_B, \forall (i_A, j_A) = \mathbf{ind}(\tilde{\mathbf{A}}), (i_B, j_B) = \mathbf{ind}(\tilde{\mathbf{B}})\}$  is created. The value of  
7450 each of its elements is computed by

$$7451 \quad T_{i_A \cdot m_B + i_B, j_A \cdot n_B + j_B} = \tilde{\mathbf{A}}(i_A, j_A) \otimes \tilde{\mathbf{B}}(i_B, j_B),$$

7452 where  $\otimes$  is the multiplicative operator specified by the `op` parameter.

7453 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 7454 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 7455 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$7456 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

7457 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
7458 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$7459 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$7460 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$7461 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

7462 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

7463 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
7464 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
7465 mask which acts as a “write mask”.

- 7466 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
7467 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$7470 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7471 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
7472 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
7473 mask are unchanged:

$$7474 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7475 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
7476 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
7477 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
7478 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
7479 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
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)</code>	<code>GrB_Vector_apply_IndexOp_Scalar(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)</code>	<code>GrB_Vector_apply_IndexOp_BOOL(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)</code>	<code>GrB_Vector_apply_IndexOp_INT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)</code>	<code>GrB_Vector_apply_IndexOp_UINT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)</code>	<code>GrB_Vector_apply_IndexOp_INT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)</code>	<code>GrB_Vector_apply_IndexOp_UINT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)</code>	<code>GrB_Vector_apply_IndexOp_INT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)</code>	<code>GrB_Vector_apply_IndexOp_UINT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)</code>	<code>GrB_Vector_apply_IndexOp_INT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)</code>	<code>GrB_Vector_apply_IndexOp_UINT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)</code>	<code>GrB_Vector_apply_IndexOp_FP32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)</code>	<code>GrB_Vector_apply_IndexOp_FP64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,<i>other</i>,...)</code>	<code>GrB_Vector_apply_IndexOp_UDT(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,const void*,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)</code>	<code>GrB_Matrix_apply_IndexOp_Scalar(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)</code>	<code>GrB_Matrix_apply_IndexOp_BOOL(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)</code>	<code>GrB_Matrix_apply_IndexOp_INT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)</code>	<code>GrB_Matrix_apply_IndexOp_UINT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)</code>	<code>GrB_Matrix_apply_IndexOp_INT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)</code>	<code>GrB_Matrix_apply_IndexOp_UINT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)</code>	<code>GrB_Matrix_apply_IndexOp_INT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)</code>	<code>GrB_Matrix_apply_IndexOp_UINT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)</code>	<code>GrB_Matrix_apply_IndexOp_INT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)</code>	<code>GrB_Matrix_apply_IndexOp_UINT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)</code>	<code>GrB_Matrix_apply_IndexOp_FP32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)</code>	<code>GrB_Matrix_apply_IndexOp_FP64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,<i>other</i>,...)</code>	<code>GrB_Matrix_apply_IndexOp_UDT(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,const void*,...)</code>

Table 5.10: Long-name, nonpolymorphic form of GraphBLAS methods (continued).[\[Scott: NEW CONTENT\]](#)

Polymorphic signature	Nonpolymorphic signature
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)</code>	<code>GrB_Vector_select_Scalar(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)</code>	<code>GrB_Vector_select_BOOL(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)</code>	<code>GrB_Vector_select_INT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)</code>	<code>GrB_Vector_select_UINT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)</code>	<code>GrB_Vector_select_INT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)</code>	<code>GrB_Vector_select_UINT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)</code>	<code>GrB_Vector_select_INT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)</code>	<code>GrB_Vector_select_UINT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)</code>	<code>GrB_Vector_select_INT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)</code>	<code>GrB_Vector_select_UINT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)</code>	<code>GrB_Vector_select_FP32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)</code>	<code>GrB_Vector_select_FP64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,other,...)</code>	<code>GrB_Vector_select_UDT(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,const void*,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)</code>	<code>GrB_Matrix_select_Scalar(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)</code>	<code>GrB_Matrix_select_BOOL(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)</code>	<code>GrB_Matrix_select_INT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)</code>	<code>GrB_Matrix_select_UINT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)</code>	<code>GrB_Matrix_select_INT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)</code>	<code>GrB_Matrix_select_UINT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)</code>	<code>GrB_Matrix_select_INT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)</code>	<code>GrB_Matrix_select_UINT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)</code>	<code>GrB_Matrix_select_INT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)</code>	<code>GrB_Matrix_select_UINT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)</code>	<code>GrB_Matrix_select_FP32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)</code>	<code>GrB_Matrix_select_FP64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,other,...)</code>	<code>GrB_Matrix_select_UDT(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,const void*,...)</code>

Table 5.11: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_reduce(GrB_Vector,...,GrB_Monoid,...)	GrB_Matrix_reduce_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_reduce(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Matrix_reduce_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_reduce(GrB_Scalar,...,GrB_Monoid,GrB_Vector,...)	GrB_Vector_reduce_Monoid_Scalar(GrB_Scalar,...,GrB_Vector,...)
GrB_reduce(GrB_Scalar,...,GrB_BinaryOp,GrB_Vector,...)	GrB_Vector_reduce_BinaryOp_Scalar(GrB_Scalar,...,GrB_Vector,...)
GrB_reduce(bool*,...,GrB_Vector,...)	GrB_Vector_reduce_BOOL(bool*,...,GrB_Vector,...)
GrB_reduce(int8_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT8(int8_t*,...,GrB_Vector,...)
GrB_reduce(uint8_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT8(uint8_t*,...,GrB_Vector,...)
GrB_reduce(int16_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT16(int16_t*,...,GrB_Vector,...)
GrB_reduce(uint16_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT16(uint16_t*,...,GrB_Vector,...)
GrB_reduce(int32_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT32(int32_t*,...,GrB_Vector,...)
GrB_reduce(uint32_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT32(uint32_t*,...,GrB_Vector,...)
GrB_reduce(int64_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT64(int64_t*,...,GrB_Vector,...)
GrB_reduce(uint64_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT64(uint64_t*,...,GrB_Vector,...)
GrB_reduce(float*,...,GrB_Vector,...)	GrB_Vector_reduce_FP32(float*,...,GrB_Vector,...)
GrB_reduce(double*,...,GrB_Vector,...)	GrB_Vector_reduce_FP64(double*,...,GrB_Vector,...)
GrB_reduce( <i>other</i> *,...,GrB_Vector,...)	GrB_Vector_reduce_UDT(void*,...,GrB_Vector,...)
GrB_reduce(GrB_Scalar,...,GrB_Monoid,GrB_Matrix,...)	GrB_Matrix_reduce_Monoid_Scalar(GrB_Scalar,...,GrB_Monoid,GrB_Matrix,...)
GrB_reduce(GrB_Scalar,...,GrB_BinaryOp,GrB_Matrix,...)	GrB_Matrix_reduce_BinaryOp_Scalar(GrB_Scalar,...,GrB_BinaryOp,GrB_Matrix,...)
GrB_reduce(bool*,...,GrB_Matrix,...)	GrB_Matrix_reduce_BOOL(bool*,...,GrB_Matrix,...)
GrB_reduce(int8_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT8(int8_t*,...,GrB_Matrix,...)
GrB_reduce(uint8_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT8(uint8_t*,...,GrB_Matrix,...)
GrB_reduce(int16_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT16(int16_t*,...,GrB_Matrix,...)
GrB_reduce(uint16_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT16(uint16_t*,...,GrB_Matrix,...)
GrB_reduce(int32_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT32(int32_t*,...,GrB_Matrix,...)
GrB_reduce(uint32_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT32(uint32_t*,...,GrB_Matrix,...)
GrB_reduce(int64_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT64(int64_t*,...,GrB_Matrix,...)
GrB_reduce(uint64_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT64(uint64_t*,...,GrB_Matrix,...)
GrB_reduce(float*,...,GrB_Matrix,...)	GrB_Matrix_reduce_FP32(float*,...,GrB_Matrix,...)
GrB_reduce(double*,...,GrB_Matrix,...)	GrB_Matrix_reduce_FP64(double*,...,GrB_Matrix,...)
GrB_reduce( <i>other</i> *,...,GrB_Matrix,...)	GrB_Matrix_reduce_UDT(void*,...,GrB_Matrix,...)
GrB_kronecker(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_kronecker_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_kronecker(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_kronecker_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_kronecker(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_kronecker_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)

Table 5.12: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Scalar_get(GrB_Scalar,...,GrB_Scalar)	GrB_Scalar_get_Scalar(GrB_Scalar,...,GrB_Scalar)
GrB_Scalar_get(GrB_Scalar,...,char*)	GrB_Scalar_get_String(GrB_Scalar,...,char*)
GrB_Scalar_get(GrB_Scalar,...,GrB_Type*)	GrB_Scalar_get_Type(GrB_Scalar,...,GrB_Type*)
GrB_Scalar_get(GrB_Scalar,...,void*)	GrB_Scalar_get_VOID(GrB_Scalar,...,void*)
GrB_Vector_get(GrB_Vector,...,GrB_Scalar)	GrB_Vector_get_Scalar(GrB_Vector,...,GrB_Scalar)
GrB_Vector_get(GrB_Vector,...,char*)	GrB_Vector_get_String(GrB_Vector,...,char*)
GrB_Vector_get(GrB_Vector,...,int*)	GrB_Matrix_get_INT32(GrB_Vector,...,int*)
GrB_Vector_get(GrB_Vector,...,GrB_Type*)	GrB_Vector_get_Type(GrB_Vector,...,GrB_Type*)
GrB_Vector_get(GrB_Vector,...,void*)	GrB_Vector_get_VOID(GrB_Vector,...,void*)
GrB_Matrix_get(GrB_Matrix,...,GrB_Scalar)	GrB_Matrix_get_Scalar(GrB_Matrix,...,GrB_Scalar)
GrB_Matrix_get(GrB_Matrix,...,char*)	GrB_Matrix_get_String(GrB_Matrix,...,char*)
GrB_Matrix_get(GrB_Matrix,...,int*)	GrB_Matrix_get_INT32(GrB_Matrix,...,int*)
GrB_Matrix_get(GrB_Matrix,...,GrB_Type*)	GrB_Matrix_get_Type(GrB_Matrix,...,GrB_Type*)
GrB_Matrix_get(GrB_Matrix,...,void*)	GrB_Matrix_get_VOID(GrB_Matrix,...,void*)
GrB_UnaryOp_get(GrB_UnaryOp,...,GrB_Scalar)	GrB_UnaryOp_get_Scalar(GrB_UnaryOp,...,GrB_Scalar)
GrB_UnaryOp_get(GrB_UnaryOp,...,char*)	GrB_UnaryOp_get_String(GrB_UnaryOp,...,char*)
GrB_UnaryOp_get(GrB_UnaryOp,...,GrB_Type*)	GrB_UnaryOp_get_Type(GrB_UnaryOp,...,GrB_Type*)
GrB_UnaryOp_get(GrB_UnaryOp,...,void*)	GrB_UnaryOp_get_VOID(GrB_UnaryOp,...,void*)
GrB_IndexUnaryOp_get(GrB_IndexUnaryOp,...,GrB_Scalar)	GrB_IndexUnaryOp_get_Scalar(GrB_IndexUnaryOp,...,GrB_Scalar)
GrB_IndexUnaryOp_get(GrB_IndexUnaryOp,...,char*)	GrB_IndexUnaryOp_get_String(GrB_IndexUnaryOp,...,char*)
GrB_IndexUnaryOp_get(GrB_IndexUnaryOp,...,GrB_Type*)	GrB_IndexUnaryOp_get_Type(GrB_IndexUnaryOp,...,GrB_Type*)
GrB_IndexUnaryOp_get(GrB_IndexUnaryOp,...,void*)	GrB_IndexUnaryOp_get_VOID(GrB_IndexUnaryOp,...,void*)
GrB_BinaryOp_get(GrB_BinaryOp,...,GrB_Scalar)	GrB_BinaryOp_get_Scalar(GrB_BinaryOp,...,GrB_Scalar)
GrB_BinaryOp_get(GrB_BinaryOp,...,char*)	GrB_BinaryOp_get_String(GrB_BinaryOp,...,char*)
GrB_BinaryOp_get(GrB_BinaryOp,...,GrB_Type*)	GrB_BinaryOp_get_Type(GrB_BinaryOp,...,GrB_Type*)
GrB_BinaryOp_get(GrB_BinaryOp,...,void*)	GrB_BinaryOp_get_VOID(GrB_BinaryOp,...,void*)
GrB_Monoid_get(GrB_Monoid,...,GrB_Scalar)	GrB_Monoid_get_Scalar(GrB_Monoid,...,GrB_Scalar)
GrB_Monoid_get(GrB_Monoid,...,char*)	GrB_Monoid_get_String(GrB_Monoid,...,char*)
GrB_Monoid_get(GrB_Monoid,...,GrB_Type*)	GrB_Monoid_get_Type(GrB_Monoid,...,GrB_Type*)
GrB_Monoid_get(GrB_Monoid,...,void*)	GrB_Monoid_get_VOID(GrB_Monoid,...,void*)
GrB_Semiring_get(GrB_Semiring,...,GrB_Scalar)	GrB_Semiring_get_Scalar(GrB_Semiring,...,GrB_Scalar)
GrB_Semiring_get(GrB_Semiring,...,char*)	GrB_Semiring_get_String(GrB_Semiring,...,char*)
GrB_Semiring_get(GrB_Semiring,...,GrB_Type*)	GrB_Semiring_get_Type(GrB_Semiring,...,GrB_Type*)
GrB_Semiring_get(GrB_Semiring,...,void*)	GrB_Semiring_get_VOID(GrB_Semiring,...,void*)
GrB_Descriptor_get(GrB_Descriptor,...,GrB_Scalar)	GrB_Descriptor_get_Scalar(GrB_Descriptor,...,GrB_Scalar)
GrB_Descriptor_get(GrB_Descriptor,...,char*)	GrB_Descriptor_get_String(GrB_Descriptor,...,char*)
GrB_Descriptor_get(GrB_Descriptor,...,int*)	GrB_Descriptor_get_INT32(GrB_Descriptor,...,int*)
GrB_Descriptor_get(GrB_Descriptor,...,GrB_Type*)	GrB_Descriptor_get_Type(GrB_Descriptor,...,GrB_Type*)
GrB_Descriptor_get(GrB_Descriptor,...,void*)	GrB_Descriptor_get_VOID(GrB_Descriptor,...,void*)
GrB_Type_get(GrB_Type,...,GrB_Scalar)	GrB_Type_get_Scalar(GrB_Type,...,GrB_Scalar)
GrB_Type_get(GrB_Type,...,char*)	GrB_Type_get_String(GrB_Type,...,char*)
GrB_Type_get(GrB_Type,...,GrB_Type*)	GrB_Type_get_Type(GrB_Type,...,GrB_Type*)
GrB_Type_get(GrB_Type,...,void*)	GrB_Type_get_VOID(GrB_Type,...,void*)
GrB_Global_get(...,GrB_Scalar)	GrB_Global_get_Scalar(...,GrB_Scalar)
GrB_Global_get(...,char*)	GrB_Global_get_String(...,char*)
GrB_Global_get(...,int*)	GrB_Global_get_INT32(...,int*)
GrB_Global_get(...,GrB_Global*)	GrB_Global_get_Global(...,GrB_Global*)
GrB_Global_get(...,void*)	GrB_Global_get_VOID(...,void*)

Table 5.13: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Scalar_set(GrB_Scalar,...,GrB_Scalar)	GrB_Scalar_set_Scalar(GrB_Scalar,...,GrB_Scalar)
GrB_Scalar_set(GrB_Scalar,...,char*)	GrB_Scalar_set_String(GrB_Scalar,...,char*)
GrB_Scalar_set(GrB_Scalar,...,void*)	GrB_Scalar_set_VOID(GrB_Scalar,...,void*)
GrB_Vector_set(GrB_Vector,...,GrB_Scalar)	GrB_Vector_set_Scalar(GrB_Vector,...,GrB_Scalar)
GrB_Vector_set(GrB_Vector,...,char*)	GrB_Vector_set_String(GrB_Vector,...,char*)
GrB_Matrix_set(GrB_Matrix,...,int)	GrB_Matrix_set_INT32(GrB_Matrix,...,int)
GrB_Vector_set(GrB_Vector,...,void*)	GrB_Vector_set_VOID(GrB_Vector,...,void*)
GrB_Matrix_set(GrB_Matrix,...,GrB_Scalar)	GrB_Matrix_set_Scalar(GrB_Matrix,...,GrB_Scalar)
GrB_Matrix_set(GrB_Matrix,...,char*)	GrB_Matrix_set_String(GrB_Matrix,...,char*)
GrB_Matrix_set(GrB_Matrix,...,int)	GrB_Matrix_set_INT32(GrB_Matrix,...,int)
GrB_Matrix_set(GrB_Matrix,...,void*)	GrB_Matrix_set_VOID(GrB_Matrix,...,void*)
GrB_UnaryOp_set(GrB_UnaryOp,...,GrB_Scalar)	GrB_UnaryOp_set_Scalar(GrB_UnaryOp,...,GrB_Scalar)
GrB_UnaryOp_set(GrB_UnaryOp,...,char*)	GrB_UnaryOp_set_String(GrB_UnaryOp,...,char*)
GrB_UnaryOp_set(GrB_UnaryOp,...,void*)	GrB_UnaryOp_set_VOID(GrB_UnaryOp,...,void*)
GrB_IndexUnaryOp_set(GrB_IndexUnaryOp,...,GrB_Scalar)	GrB_IndexUnaryOp_set_Scalar(GrB_IndexUnaryOp,...,GrB_Scalar)
GrB_IndexUnaryOp_set(GrB_IndexUnaryOp,...,char*)	GrB_IndexUnaryOp_set_String(GrB_IndexUnaryOp,...,char*)
GrB_IndexUnaryOp_set(GrB_IndexUnaryOp,...,void*)	GrB_IndexUnaryOp_set_VOID(GrB_IndexUnaryOp,...,void*)
GrB_BinaryOp_set(GrB_BinaryOp,...,GrB_Scalar)	GrB_BinaryOp_set_Scalar(GrB_BinaryOp,...,GrB_Scalar)
GrB_BinaryOp_set(GrB_BinaryOp,...,char*)	GrB_BinaryOp_set_String(GrB_BinaryOp,...,char*)
GrB_BinaryOp_set(GrB_BinaryOp,...,void*)	GrB_BinaryOp_set_VOID(GrB_BinaryOp,...,void*)
GrB_Monoid_set(GrB_Monoid,...,GrB_Scalar)	GrB_Monoid_set_Scalar(GrB_Monoid,...,GrB_Scalar)
GrB_Monoid_set(GrB_Monoid,...,char*)	GrB_Monoid_set_String(GrB_Monoid,...,char*)
GrB_Monoid_set(GrB_Monoid,...,void*)	GrB_Monoid_set_VOID(GrB_Monoid,...,void*)
GrB_Semiring_set(GrB_Semiring,...,GrB_Scalar)	GrB_Semiring_set_Scalar(GrB_Semiring,...,GrB_Scalar)
GrB_Semiring_set(GrB_Semiring,...,char*)	GrB_Semiring_set_String(GrB_Semiring,...,char*)
GrB_Semiring_set(GrB_Semiring,...,void*)	GrB_Semiring_set_VOID(GrB_Semiring,...,void*)
GrB_Descriptor_set(GrB_Descriptor,...,GrB_Scalar)	GrB_Descriptor_set_Scalar(GrB_Descriptor,...,GrB_Scalar)
GrB_Descriptor_set(GrB_Descriptor,...,char*)	GrB_Descriptor_set_String(GrB_Descriptor,...,char*)
GrB_Descriptor_set(GrB_Descriptor,...,int)	GrB_Descriptor_set_INT32(GrB_Descriptor,...,int)
GrB_Descriptor_set(GrB_Descriptor,...,void*)	GrB_Descriptor_set_VOID(GrB_Descriptor,...,void*)
GrB_Type_set(GrB_Type,...,GrB_Scalar)	GrB_Type_set_Scalar(GrB_Type,...,GrB_Scalar)
GrB_Type_set(GrB_Type,...,char*)	GrB_Type_set_String(GrB_Type,...,char*)
GrB_Type_set(GrB_Type,...,void*)	GrB_Type_set_VOID(GrB_Type,...,void*)
GrB_Global_set(...,GrB_Scalar)	GrB_Global_set_Scalar(...,GrB_Scalar)
GrB_Global_set(...,char*)	GrB_Global_set_String(...,char*)
GrB_Global_set(...,int)	GrB_Global_set_INT32(...,int)
GrB_Global_set(...,void*)	GrB_Global_set_VOID(...,void*)





# Appendix A

## Revision history

Changes in 2.0.1 (Released: ## Xxxxx 2022:

- (Issue GH-69) Fix error in description of contents of matrix constructed from GrB\_Matrix\_diag.

Changes in 2.0.0 (Released: 15 November 2021:

- Reorganized Chapters 2 and 3: Chapter 2 contains prose regarding the basic concepts captured in the API; Chapter 3 presents all of the enumerations, literals, data types, and predefined objects required by the API. Made short captions for the List of Tables.
- (Issue BB-49, BB-50) Updated and corrected language regarding multithreading and completion, and requirements regarding acquire-release memory orders. Methods that used to force complete no longer do.
- (Issue BB-74, BB-9) Assigned integer values to all return codes as well as all enumerations in the API to ensure run-time compatibility between libraries.
- (Issues BB-70, BB-67) Changed semantics and signature of GrB\_wait(obj, mode). Added wait modes for 'complete' or 'materialize' and removed GrB\_wait(void). **This breaks backward compatibility.**
- (Issue GH-51) Removed deprecated GrB\_SCMP literal from descriptor values. **This breaks backward compatibility.**
- (Issues BB-8, BB-36) Added sparse GrB\_Scalar object and its use in additional variants of extract/setElement methods, and reduce, apply, assign and select operations.
- (Issues BB-34, GH-33, GH-45) Added new select operation that uses an index unary operator. Added new variants of apply that take an index unary operator (matrix and vector variants).
- (Issues BB-68, BB-51) Added serialize and deserialize methods for matrices to/from implementation defined formats.

- 7510 • (Issues BB-25, GH-42) Added import and export methods for matrices to/from API specified  
7511 formats. Three formats have been specified: CSC, CSR, COO. Dense row and column formats  
7512 have been deferred.
- 7513 • (Issue BB-75) Added matrix constructor to build a diagonal `GrB_Matrix` from a `GrB_Vector`.
- 7514 • (Issue BB-73) Allow `GrB_NULL` for dup operator in matrix and vector `build` methods. Return  
7515 error if duplicate locations encountered.
- 7516 • (Issue BB-58) Added matrix and vector methods to remove (annihilate) elements.
- 7517 • (Issue BB-17) Added `GrB_ABS_T` (absolute value) unary operator.
- 7518 • (Issue GH-46) Adding `GrB_ONEB_T` binary operator that returns 1 cast to type T (not to  
7519 be confused with the proposed unary operator).
- 7520 • (Issue GH-53) Added language about what constitutes a “conformant” implementation. Added  
7521 `GrB_NOT_IMPLEMENTED` return value (API error) for API any combinations of inputs to  
7522 a method that is not supported by the implementation.
- 7523 • Added `GrB_EMPTY_OBJECT` return value (execution error) that is used when an opaque  
7524 object (currently only `GrB_Scalar`) is passed as an input that cannot be empty.
- 7525 • (Issue BB-45) Removed language about annihilators.
- 7526 • (Issue BB-69) Made names/symbols containing underscores searchable in PDF.
- 7527 • Updated a number algorithms in the appendix to use new operations and methods.
- 7528 • Numerous additions (some changes) to the non-polymorphic interface to track changes to the  
7529 specification.
- 7530 • Typographical error in version macros was corrected. They are all caps: `GRB_VERSION` and  
7531 `GRB_SUBVERSION`.
- 7532 • Typographical change to `eWiseAdd` Description to be consistent in order of set intersections.
- 7533 • Typographical errors in `eWiseAdd`: cut-and-paste errors from `eWiseMult`/set intersection  
7534 fixed to read `eWiseAdd`/set union.
- 7535 • Typographical error (`NEQ`  $\rightarrow$  `NE`) in Description of Table 3.8.

7536 Changes in 1.3.0 (Released: 25 September 2019):

- 7537 • (Issue BB-50) Changed definition of completion and added `GrB_wait()` that takes an opaque  
7538 GraphBLAS object as an argument.
- 7539 • (Issue BB-39) Added `GrB_kronecker` operation.
- 7540 • (Issue BB-40) Added variants of the `GrB_apply` operation that take a binary function and a  
7541 scalar.

7542  
7543  
7544  
7545  
7546  
7547  
7548  
7549  
7550  
7551  
7552  
7553  
7554  
7555  
7556  
7557  
7558  
7559  
7560  
7561  
7562  
7563  
7564  
7565  
7566  
7567  
7568  
7569  
7570  
7571  
7572  
7573  
7574

- (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.

7575 • Fixed miscellaneous typographical errors (such as  $\otimes$ ,  $\oplus$ ).

7576 Changes in 1.2.0:

7577 • Removed "provisional" clause.

7578 Changes in 1.1.0:

7579 • Removed unnecessary `const` from `nindices`, `nrows`, and `ncols` parameters of both `extract` and  
7580 `assign` operations.

7581 • Signature of `GrB_UnaryOp_new` changed: order of input parameters changed.

7582 • Signature of `GrB_BinaryOp_new` changed: order of input parameters changed.

7583 • Signature of `GrB_Monoid_new` changed: removal of domain argument which is now inferred  
7584 from the domains of the binary operator provided.

7585 • Signature of `GrB_Vector_extractTuples` and `GrB_Matrix_extractTuples` to add an in/out ar-  
7586 gument, `n`, which indicates the size of the output arrays provided (in terms of number of  
7587 elements, not number of bytes). Added new execution error, `GrB_INSUFFICIENT_SPACE`  
7588 which is returned when the capacities of the output arrays are insufficient to hold all of the  
7589 tuples.

7590 • Changed `GrB_Column_assign` to `GrB_Col_assign` for consistency in non-polymorphic inter-  
7591 face.

7592 • Added replace flag (`z`) notation to Table 4.1.

7593 • Updated the "Mathematical Description" of the `assign` operation in Table 4.1.

7594 • Added triangle counting example.

7595 • Added subsection headers for `accumulate` and `mask/replace` discussions in the Description  
7596 sections of GraphBLAS operations when the respective text was the "standard" text (i.e.,  
7597 identical in a majority of the operations).

7598 • Fixed typographical errors.

7599 Changes in 1.0.2:

7600 • Expanded the definitions of `Vector_build` and `Matrix_build` to conceptually use intermediate  
7601 matrices and avoid casting issues in certain implementations.

7602 • Fixed the bug in the `GrB_assign` definition. Elements of the output object are no longer being  
7603 erased outside the assigned area.

7604 • Changes non-polymorphic interface:

7605 – Renamed `GrB_Row_extract` to `GrB_Col_extract`.

- 7606           – Renamed GrB\_Vector\_reduce\_BinaryOp to GrB\_Matrix\_reduce\_BinaryOp.
- 7607           – Renamed GrB\_Vector\_reduce\_Monoid to GrB\_Matrix\_reduce\_Monoid.
- 7608       • Fixed the bugs with respect to isolated vertices in the Maximal Independent Set example.
- 7609       • Fixed numerous typographical errors.



## Appendix B

# Non-opaque data format definitions

### B.1 GrB\_Format: Specify the format for input/output of a GraphBLAS matrix.

In this section, the non-opaque matrix formats specified by GrB\_Format and used in matrix import and export methods are defined.

#### B.1.1 GrB\_CSR\_FORMAT

The GrB\_CSR\_FORMAT format indicates that a matrix will be imported or exported using the compressed sparse row (CSR) format. `indptr` is a pointer to an array of `GrB_Index` of size `nrows+1` elements, where the `i`'th index will contain the starting index in the `values` and `indices` arrays corresponding to the `i`'th row of the matrix. `indices` is a pointer to an array of number of stored elements (each a `GrB_Index`), where each element contains the corresponding element's column index within a row of the matrix. `values` is a pointer to an array of number of stored elements (each the size of the scalar stored in the matrix) containing the corresponding value. The elements of each row are not required to be sorted by column index.

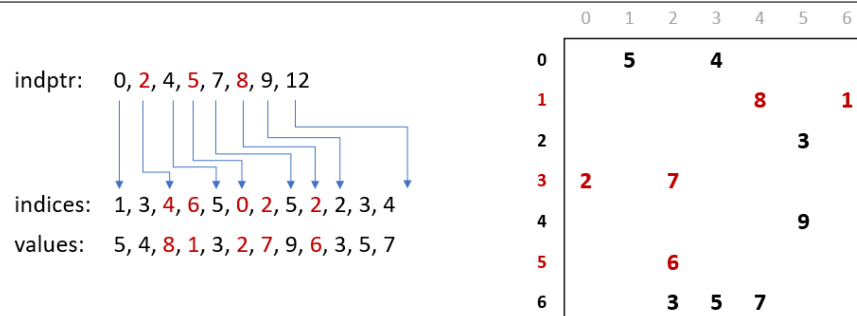


Figure B.1: Data layout for CSR format.

### B.1.2 GrB\_CSC\_FORMAT

The GrB\_CSC\_FORMAT format indicates that a matrix will be imported or exported using the compressed sparse column (CSC) format. `indptr` is a pointer to an array of `GrB_Index` of size `ncols+1` elements, where the *i*'th index will contain the starting index in the `values` and `indices` arrays corresponding to the *i*'th column of the matrix. `indices` is a pointer to an array of number of stored elements (each a `GrB_Index`), where each element contains the corresponding element's row index within a column of the matrix. `values` is a pointer to an array of number of stored elements (each the size of the scalar stored in the matrix) containing the corresponding value. The elements of each column are not required to be sorted by row index.

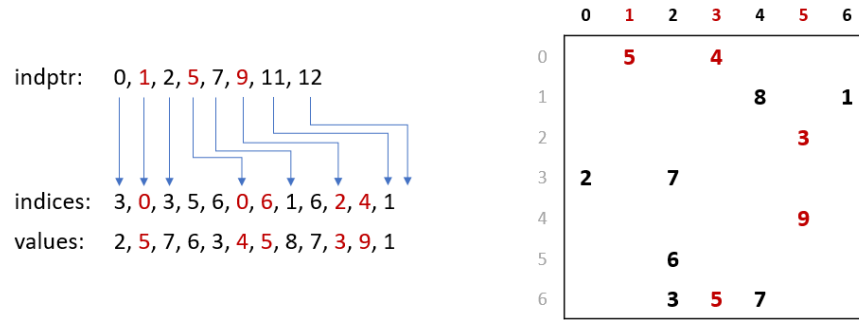


Figure B.2: Data layout for CSC format.

### B.1.3 GrB\_COO\_FORMAT

The GrB\_COO\_FORMAT format indicates that a matrix will be imported or exported using the coordinate list (COO) format. `indptr` is a pointer to an array of `GrB_Index` of size number of stored elements, where each element contains the corresponding element's column index. `indices` will be a pointer to an array of `GrB_Index` of size number of stored elements, where each element contains the corresponding element's row index. `values` will be a pointer to an array of size number of stored elements (each the size of the scalar stored in the matrix) containing the corresponding value. Elements are not required to be sorted in any order.

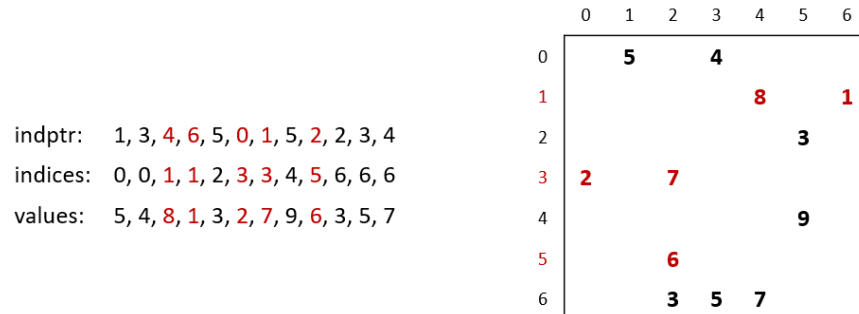


Figure B.3: Data layout for COO format.



7642 **Appendix C**

7643 **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) = d^th level frontier from source vertex s
43         GrB_Matrix_new(&(sigmas[d]), GrB_BOOL, n, nsver);
44
45         GrB_apply(sigmas[d], GrB_NULL, GrB_NULL,
46                 GrB_IDENTITY_BOOL, frontier, GrB_NULL); // sigmas[d](:,:) = (Boolean) frontier
47         GrB_eWiseAdd(numsp, GrB_NULL, GrB_NULL, GrB_PLUS_INT32,
48                     numsp, frontier, GrB_NULL); // numsp += frontier (accum path counts)
49         GrB_mxm(frontier, numsp, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
50                 A, frontier, GrB_DESC_RCT0); // f<!numsp> = A' +.* f (update frontier)
51         GrB_Matrix_nvals(&nvals, frontier); // number of nodes in frontier at this level
52         d++;
53     } while (nvals);
54
55     // nspinv: the inverse of the number of shortest paths for each node and starting vertex.
56     GrB_Matrix nspinv;
57     GrB_Matrix_new(&nspinv, GrB_FP32, n, nsver);
58     GrB_apply(nspinv, GrB_NULL, GrB_NULL,
59              GrB_MINV_FP32, numsp, GrB_NULL); // nspinv = 1./numsp
60
61     // bcu: BC updates for each vertex for each starting vertex in s
62     GrB_Matrix bcu;

```

```

63 GrB_Matrix_new(&bcu, GrB_FP32, n, nsver);
64 GrB_assign(bcu, GrB_NULL, GrB_NULL,
65           1.0f, GrB_ALL, n, GrB_ALL, nsver, GrB_NULL); // filled with 1 to avoid sparsity issues
66
67 GrB_Matrix w; // temporary workspace matrix
68 GrB_Matrix_new(&w, GrB_FP32, n, nsver);
69
70 // ----- Tally phase (backward sweep) -----
71 for (int i=d-1; i>0; i--) {
72     GrB_eWiseMult(w, sigmas[i], GrB_NULL,
73                 GrB_TIMES_FP32, bcu, nspinv, GrB_DESC_R); // w<sigmas[i]>=(1 ./ nsp).*bcu
74
75     // add contributions by successors and mask with that BFS level's frontier
76     GrB_mxm(w, sigmas[i-1], GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32,
77            A, w, GrB_DESC_R); // w<sigmas[i-1]> = (A +.* w)
78     GrB_eWiseMult(bcu, GrB_NULL, GrB_PLUS_FP32, GrB_TIMES_FP32,
79                 w, numsp, GrB_NULL); // bcu += w .* numsp
80 }
81
82 // row reduce bcu and subtract "nsver" from every entry to account
83 // for 1 extra value per bcu row element.
84 GrB_reduce(*delta, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, bcu, GrB_NULL);
85 GrB_apply(*delta, GrB_NULL, GrB_NULL, GrB_MINUS_FP32, *delta, (float)nsver, GrB_NULL);
86
87 // Release resources
88 for (int i=0; i<d; i++) {
89     GrB_free(&(sigmas[i]));
90 }
91 free(sigmas);
92
93 GrB_free(&frontier); GrB_free(&numsp);
94 GrB_free(&nspinv); GrB_free(&bcu); GrB_free(&w);
95
96 return GrB_SUCCESS;
97 }

```



## C.6 Example: Maximal independent set (MIS) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  // Assign a random number to each element scaled by the inverse of the node's degree.
8  // This will increase the probability that low degree nodes are selected and larger
9  // sets are selected.
10 void setRandom(void *out, const void *in)
11 {
12     uint32_t degree = *(uint32_t*)in;
13     *(float*)out = (0.0001f + random()/(1. + 2.*degree)); // add 1 to prevent divide by zero
14 }
15
16 /*
17  * A variant of Luby's randomized algorithm [Luby 1985].
18  *
19  * Given a numeric n x n adjacency matrix A of an unweighted and undirected graph (where
20  * the value true represents an edge), compute a maximal set of independent vertices and
21  * return it in a boolean n-vector, 'iset' where set[i] == true implies vertex i is a member
22  * of the set (the iset vector should be uninitialized on input.)
23  */
24 GrB_Info MIS(GrB_Vector *iset, const GrB_Matrix A)
25 {
26     GrB_Index n;
27     GrB_Matrix_nrows(&n,A); // n = # of rows of A
28
29     GrB_Vector prob; // holds random probabilities for each node
30     GrB_Vector neighbor_max; // holds value of max neighbor probability
31     GrB_Vector new_members; // holds set of new members to iset
32     GrB_Vector new_neighbors; // holds set of new neighbors to new iset mbrs.
33     GrB_Vector candidates; // candidate members to iset
34
35     GrB_Vector_new(&prob,GrB_FP32,n);
36     GrB_Vector_new(&neighbor_max,GrB_FP32,n);
37     GrB_Vector_new(&new_members,GrB_BOOL,n);
38     GrB_Vector_new(&new_neighbors,GrB_BOOL,n);
39     GrB_Vector_new(&candidates,GrB_BOOL,n);
40     GrB_Vector_new(iset,GrB_BOOL,n); // Initialize independent set vector, bool
41
42     GrB_UnaryOp set_random;
43     GrB_UnaryOp_new(&set_random,setRandom,GrB_FP32,GrB_UINT32);
44
45     // compute the degree of each vertex.
46     GrB_Vector degrees;
47     GrB_Vector_new(&degrees,GrB_FP64,n);
48     GrB_reduce(degrees,GrB_NULL,GrB_NULL,GrB_PLUS_FP64,A,GrB_NULL);
49
50     // Isolated vertices are not candidates: candidates[degrees != 0] = true
51     GrB_assign(candidates,degrees,GrB_NULL,true,GrB_ALL,n,GrB_NULL);
52
53     // add all singletons to iset: iset[degree == 0] = 1
54     GrB_assign(*iset,degrees,GrB_NULL,true,GrB_ALL,n,GrB_DESC_RC);
55
56     // Iterate while there are candidates to check.
57     GrB_Index nvals;
58     GrB_Vector_nvals(&nvals,candidates);
59     while (nvals > 0) {
60         // compute a random probability scaled by inverse of degree
61         GrB_apply(prob,candidates,GrB_NULL,set_random,degrees,GrB_DESC_R);
62     }

```

```

63 // compute the max probability of all neighbors
64 GrB_mnv(neighbor_max, candidates, GrB_NULL, GrB_MAX_SECOND_SEMIRING_FP32, A, prob, GrB_DESC_R);
65
66 // select vertex if its probability is larger than all its active neighbors,
67 // and apply a "masked no-op" to remove stored falses
68 GrB_eWiseAdd(new_members, GrB_NULL, GrB_NULL, GrB_GT_FP64, prob, neighbor_max, GrB_NULL);
69 GrB_apply(new_members, new_members, GrB_NULL, GrB_IDENTITY_BOOL, new_members, GrB_DESC_R);
70
71 // add new members to independent set.
72 GrB_eWiseAdd(*iset, GrB_NULL, GrB_NULL, GrB_LOR, *iset, new_members, GrB_NULL);
73
74 // remove new members from set of candidates  $c = c \& \neg \text{new}$ 
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 }
```