

# 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 2022/11/14 at 11:46:38 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	3.7	Descriptors . . . . .	48
60	3.8	GrB_Info return values . . . . .	48
61	<b>4</b>	<b>Methods</b>	<b>53</b>
62	4.1	Context methods . . . . .	53
63	4.1.1	init: Initialize a GraphBLAS context . . . . .	53
64	4.1.2	finalize: Finalize a GraphBLAS context . . . . .	54
65	4.1.3	getVersion: Get the version number of the standard. . . . .	55
66	4.2	Object methods . . . . .	55
67	4.2.1	Query methods . . . . .	56
68	4.2.1.1	get: Query the value of an object . . . . .	56
69	4.2.1.2	set: Set field of an object . . . . .	57
70	4.2.2	Algebra methods . . . . .	58
71	4.2.2.1	Type_new: Construct a new GraphBLAS (user-defined) type . . . .	58
72	4.2.2.2	UnaryOp_new: Construct a new GraphBLAS unary operator . . . .	59
73	4.2.2.3	BinaryOp_new: Construct a new GraphBLAS binary operator . . .	60
74	4.2.2.4	Monoid_new: Construct a new GraphBLAS monoid . . . . .	62
75	4.2.2.5	Semiring_new: Construct a new GraphBLAS semiring . . . . .	63
76	4.2.2.6	IndexUnaryOp_new: Construct a new GraphBLAS index unary op-	
77		erator [Scott: NEW CONTENT] . . . . .	64

78	4.2.3	Scalar methods . . . . .	66
79	4.2.3.1	Scalar_new: Construct a new scalar . . . . .	66
80	4.2.3.2	Scalar_dup: Construct a copy of a GraphBLAS scalar . . . . .	67
81	4.2.3.3	Scalar_clear: Clear/remove a stored value from a scalar . . . . .	68
82	4.2.3.4	Scalar_nvals: Number of stored elements in a scalar . . . . .	69
83	4.2.3.5	Scalar_setElement: Set the single element in a scalar . . . . .	70
84	4.2.3.6	Scalar_extractElement: Extract a single element from a scalar. . . .	71
85	4.2.4	Vector methods . . . . .	73
86	4.2.4.1	Vector_new: Construct new vector . . . . .	73
87	4.2.4.2	Vector_dup: Construct a copy of a GraphBLAS vector . . . . .	74
88	4.2.4.3	Vector_resize: Resize a vector . . . . .	75
89	4.2.4.4	Vector_clear: Clear a vector . . . . .	76
90	4.2.4.5	Vector_size: Size of a vector . . . . .	77
91	4.2.4.6	Vector_nvals: Number of stored elements in a vector . . . . .	77
92	4.2.4.7	Vector_build: Store elements from tuples into a vector . . . . .	78
93	4.2.4.8	Vector_setElement: Set a single element in a vector . . . . .	80
94	4.2.4.9	Vector_removeElement: Remove an element from a vector . . . . .	82
95	4.2.4.10	Vector_extractElement: Extract a single element from a vector. . . .	83
96	4.2.4.11	Vector_extractTuples: Extract tuples from a vector . . . . .	85
97	4.2.5	Matrix methods . . . . .	86
98	4.2.5.1	Matrix_new: Construct new matrix . . . . .	86
99	4.2.5.2	Matrix_dup: Construct a copy of a GraphBLAS matrix . . . . .	88
100	4.2.5.3	Matrix_diag: Construct a diagonal GraphBLAS matrix . . . . .	89
101	4.2.5.4	Matrix_resize: Resize a matrix . . . . .	90
102	4.2.5.5	Matrix_clear: Clear a matrix . . . . .	91
103	4.2.5.6	Matrix_nrows: Number of rows in a matrix . . . . .	92
104	4.2.5.7	Matrix_ncols: Number of columns in a matrix . . . . .	92
105	4.2.5.8	Matrix_nvals: Number of stored elements in a matrix . . . . .	93
106	4.2.5.9	Matrix_build: Store elements from tuples into a matrix . . . . .	94
107	4.2.5.10	Matrix_setElement: Set a single element in matrix . . . . .	96

108	4.2.5.11	Matrix_removeElement: Remove an element from a matrix . . . . .	98
109	4.2.5.12	Matrix_extractElement: Extract a single element from a matrix . . .	99
110	4.2.5.13	Matrix_extractTuples: Extract tuples from a matrix . . . . .	101
111	4.2.5.14	Matrix_exportHint: Provide a hint as to which storage format might	
112		be most efficient for exporting a matrix . . . . .	103
113	4.2.5.15	Matrix_exportSize: Return the array sizes necessary to export a	
114		GraphBLAS matrix object . . . . .	104
115	4.2.5.16	Matrix_export: Export a GraphBLAS matrix to a pre-defined format	105
116	4.2.5.17	Matrix_import: Import a matrix into a GraphBLAS object . . . . .	107
117	4.2.5.18	Matrix_serializeSize: Compute the serialize buffer size . . . . .	109
118	4.2.5.19	Matrix_serialize: Serialize a GraphBLAS matrix. . . . .	110
119	4.2.5.20	Matrix_deserialize: Deserialize a GraphBLAS matrix. . . . .	111
120	4.2.6	Descriptor methods . . . . .	112
121	4.2.6.1	Descriptor_new: Create new descriptor . . . . .	112
122	4.2.6.2	Descriptor_set: Set content of descriptor . . . . .	113
123	4.2.7	free: Destroy an object and release its resources . . . . .	114
124	4.2.8	wait: Return once an object is either <i>complete</i> or <i>materialized</i> . . . . .	116
125	4.2.9	error: Retrieve an error string . . . . .	117
126	4.3	GraphBLAS operations . . . . .	118
127	4.3.1	mxm: Matrix-matrix multiply . . . . .	122
128	4.3.2	vxm: Vector-matrix multiply . . . . .	127
129	4.3.3	mxv: Matrix-vector multiply . . . . .	131
130	4.3.4	eWiseMult: Element-wise multiplication . . . . .	135
131	4.3.4.1	eWiseMult: Vector variant . . . . .	136
132	4.3.4.2	eWiseMult: Matrix variant . . . . .	140
133	4.3.5	eWiseAdd: Element-wise addition . . . . .	145
134	4.3.5.1	eWiseAdd: Vector variant . . . . .	146
135	4.3.5.2	eWiseAdd: Matrix variant . . . . .	150
136	4.3.6	extract: Selecting sub-graphs . . . . .	156
137	4.3.6.1	extract: Standard vector variant . . . . .	156
138	4.3.6.2	extract: Standard matrix variant . . . . .	160

139	4.3.6.3	extract: Column (and row) variant . . . . .	165
140	4.3.7	assign: Modifying sub-graphs . . . . .	170
141	4.3.7.1	assign: Standard vector variant . . . . .	170
142	4.3.7.2	assign: Standard matrix variant . . . . .	175
143	4.3.7.3	assign: Column variant . . . . .	181
144	4.3.7.4	assign: Row variant . . . . .	186
145	4.3.7.5	assign: Constant vector variant[Scott: NEW CONTENT] . . . . .	192
146	4.3.7.6	assign: Constant matrix variant[Scott: NEW CONTENT] . . . . .	197
147	4.3.8	apply: Apply a function to the elements of an object . . . . .	203
148	4.3.8.1	apply: Vector variant . . . . .	203
149	4.3.8.2	apply: Matrix variant . . . . .	208
150	4.3.8.3	apply: Vector-BinaryOp variants[Scott: NEW CONTENT] . . . . .	212
151	4.3.8.4	apply: Matrix-BinaryOp variants[Scott: NEW CONTENT] . . . . .	218
152	4.3.8.5	apply: Vector index unary operator variant[Scott: NEW CONTENT] . . . . .	224
153	4.3.8.6	apply: Matrix index unary operator variant[Scott: NEW CONTENT] . . . . .	229
154	4.3.9	select: . . . . .	234
155	4.3.9.1	select: Vector variant[Scott: NEW CONTENT] . . . . .	234
156	4.3.9.2	select: Matrix variant[Scott: NEW CONTENT] . . . . .	239
157	4.3.10	reduce: Perform a reduction across the elements of an object . . . . .	245
158	4.3.10.1	reduce: Standard matrix to vector variant . . . . .	245
159	4.3.10.2	reduce: Vector-scalar variant[Scott: NEW CONTENT] . . . . .	249
160	4.3.10.3	reduce: Matrix-scalar variant[Scott: NEW CONTENT] . . . . .	253
161	4.3.11	transpose: Transpose rows and columns of a matrix . . . . .	256
162	4.3.12	kroncker: Kronecker product of two matrices . . . . .	260
163	<b>5</b>	<b>Nonpolymorphic interface[Scott: NEW CONTENT]</b>	<b>267</b>
164	<b>A</b>	<b>Revision history</b>	<b>279</b>
165	<b>B</b>	<b>Non-opaque data format definitions</b>	<b>285</b>
166	B.1	GrB_Format: Specify the format for input/output of a GraphBLAS matrix. . . . .	285
167	B.1.1	GrB_CSR_FORMAT . . . . .	285

168	B.1.2 GrB_CSC_FORMAT . . . . .	286
169	B.1.3 GrB_COO_FORMAT . . . . .	286
170	<b>C Examples</b>	<b>287</b>
171	C.1 Example: Level breadth-first search (BFS) in GraphBLAS . . . . .	288
172	C.2 Example: Level BFS in GraphBLAS using apply . . . . .	289
173	C.3 Example: Parent BFS in GraphBLAS . . . . .	290
174	C.4 Example: Betweenness centrality (BC) in GraphBLAS . . . . .	291
175	C.5 Example: Batched BC in GraphBLAS . . . . .	293
176	C.6 Example: Maximal independent set (MIS) in GraphBLAS . . . . .	295
177	C.7 Example: Counting triangles in GraphBLAS . . . . .	297



# List of Tables

178		
179	2.1	Types of GraphBLAS opaque objects. . . . . 23
180	2.2	Methods that forced completion prior to GraphBLAS v2.0. . . . . 28
181	3.1	Enumeration literals and corresponding values input to various GraphBLAS methods. 32
182	3.2	Predefined GrB_Type values. . . . . 33
183	3.3	Operator input for relevant GraphBLAS operations. . . . . 34
184	3.4	Properties and recipes for building GraphBLAS algebraic objects. . . . . 35
185	3.5	Predefined unary and binary operators for GraphBLAS in C. . . . . 37
186	3.6	Predefined index unary operators for GraphBLAS in C. . . . . 38
187	3.7	Predefined monoids for GraphBLAS in C. . . . . 40
188	3.8	Predefined “true” semirings for GraphBLAS in C. . . . . 41
189	3.9	Other useful predefined semirings for GraphBLAS in C. . . . . 42
190	3.10	GrB_Format enumeration literals and corresponding values for matrix import and
191		export methods. . . . . 44
192	3.11	Field values of type GrB_Field enumeration, corresponding types, and the objects
193		which must implement that GrB_Field. Collection refers to GrB_Matrix, GrB_Vector,
194		and GrB_Scalar, Algebraic refers to Operators, Monoids, and Semirings, while All refers
195		to all GraphBLAS objects. Global fields are denoted by Global. All fields may be
196		read, some may be written (denoted by W), and some are hints (denoted by H) which
197		may be ignored by the implementation. . . . . 47
198	3.12	Descriptor types and literals for fields and values. . . . . 49
199	3.13	Predefined GraphBLAS descriptors. . . . . 50
200	3.14	Enumeration literals and corresponding values returned by GraphBLAS methods
201		and operations. . . . . 51
202	4.1	A mathematical notation for the fundamental GraphBLAS operations supported in
203		this specification. . . . . 119

204	5.1	Long-name, nonpolymorphic form of GraphBLAS methods. . . . .	267
205	5.2	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	268
206	5.3	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	269
207	5.4	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	270
208	5.5	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	271
209	5.6	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	272
210	5.7	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	273
211	5.8	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	274
212	5.9	Long-name, nonpolymorphic form of GraphBLAS methods (continued).[Scott: NEW	
213		CONTENT] . . . . .	275
214	5.10	Long-name, nonpolymorphic form of GraphBLAS methods (continued).[Scott: NEW	
215		CONTENT] . . . . .	276
216	5.11	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	277

# 217 List of Figures

218	3.1 Hierarchy of algebraic object classes in GraphBLAS. . . . .	43
219	4.1 Flowchart for the GraphBLAS operations. . . . .	120
220	B.1 Data layout for CSR format. . . . .	285
221	B.2 Data layout for CSC format. . . . .	286
222	B.3 Data layout for COO format. . . . .	286

## Acknowledgments

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

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

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

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

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

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

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

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

# Chapter 1

## Introduction

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

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

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

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

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

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

is not possible to express the signature of that method.

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

The remainder of this document is organized as follows:

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

## Chapter 2

# Basic concepts

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

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

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

## 2.1 Glossary

### 2.1.1 GraphBLAS API basic definitions

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

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

### 2.1.2 GraphBLAS objects and their structure

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



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

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

### 2.1.3 Algebraic structures used in the GraphBLAS

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

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

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

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

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

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

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

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

### 2.1.5 GraphBLAS methods: behaviors and error conditions

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

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

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

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

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

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

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

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

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

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

### 788 3.3 Types (domains)

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

---

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

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

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

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

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

---



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

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

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

### 3.4 Algebraic objects, operators and associated functions

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

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

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

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

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

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

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

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

### 3.4.1 Operators

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

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

---

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

(a) Properties of algebraic objects.

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

(b) Recipes for algebraic objects.

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

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

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

---

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

837 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  
838 domains,  $D_{out}$ ,  $D_{in_1}$ ,  $D_{in_2}$ , the domain of GraphBLAS indices, and an operation  $f_i : D_{in_1} \times I_{U64}^2 \times$   
839  $D_{in_2} \rightarrow D_{out}$  (where  $I_{U64}$  corresponds to the domain of a `GrB_Index`). For a given GraphBLAS  
840 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  $\mathbf{f}(F_i) = f_i$ .

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

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

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

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

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

### 3.4.2 Monoids

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

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

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

### 3.4.3 Semirings

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

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

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

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

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

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

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

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

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



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

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

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

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

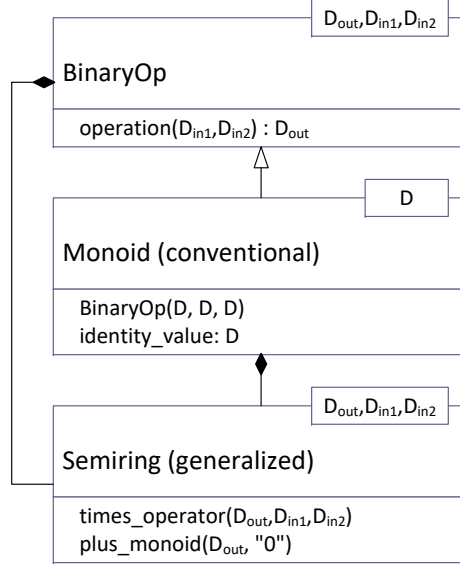


Figure 3.1: Hierarchy of algebraic object classes in GraphBLAS. GraphBLAS semirings consist of a conventional monoid with one domain for the addition function, and a binary operator with three domains for the multiplication function.

## 3.5 Collections

### 3.5.1 Scalars

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

### 3.5.2 Vectors

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

### 3.5.3 Matrices

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

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

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

#### 3.5.3.1 External matrix formats

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

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

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

### 3.5.4 Masks

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

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

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

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

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

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

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

### 3.6 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

963 by other operations, such as `GrB_BLOCKING_MODE` which is determined by `GrB_Init`.  
964 Several fields are only *hints*. A GraphBLAS implementation is free to ignore a hint and return  
965 `GrB_SUCCESS`. For instance `GrB_NTHREADS` might be ignored by a sequential GraphBLAS im-  
966 plementation.

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

972 String handling works as follows: strings are passed as null-terminated `char*` arrays to `set`, which  
973 are then copied into internal data structures. On output the caller of `get` passes a preallocated  
974 `char*` whose size is determined by another field (`GrB_NAMESIZE`, for instance) or is a constant.  
975 The internal null-terminated string is then copied into this buffer, including the null character.

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.

(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.

(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`.

## 1031 **Return Values**

1032                   GrB\_SUCCESS operation completed successfully.

1033                   GrB\_PANIC unknown internal error.

1034                   GrB\_INVALID\_VALUE invalid mode specified, or method called multiple times.

## 1035 **Description**

1036 The init method creates and initializes a GraphBLAS C API context. The argument to GrB\_init  
1037 defines the mode for the context. The two available modes are:

- 1038       • GrB\_BLOCKING: In this mode, each method in a sequence returns after its computations have  
1039        completed and output arguments are available to subsequent statements in an application.  
1040        When executing in GrB\_BLOCKING mode, the methods execute in program order.
- 1041       • GrB\_NONBLOCKING: In this mode, methods in a sequence may return after arguments in  
1042        the method have been tested for dimension and domain compatibility within the method  
1043        but potentially before their computations complete. Output arguments are available to sub-  
1044        sequent GraphBLAS methods in an application. When executing in GrB\_NONBLOCKING  
1045        mode, the methods in a sequence may execute in any order that preserves the mathematical  
1046        result defined by the sequence.

1047 An application can only create one context per execution instance. An application may only call  
1048 GrB\_Init once. Calling GrB\_Init more than once results in undefined behavior.

## 1049 **4.1.2 finalize: Finalize a GraphBLAS context**

1050 Terminates and frees any internal resources created to support the GraphBLAS C API context.

## 1051 **C Syntax**

1052                   GrB\_Info GrB\_finalize();

## 1053 **Return Values**

1054                   GrB\_SUCCESS operation completed successfully.

1055                   GrB\_PANIC unknown internal error.

## 1056 **Description**

1057 The `finalize` method terminates and frees any internal resources created to support the GraphBLAS  
1058 C API context. `GrB_finalize` may only be called after a context has been initialized by calling  
1059 `GrB_init`, or else undefined behavior occurs. After `GrB_finalize` has been called to finalize a Graph-  
1060 BLAS context, calls to any GraphBLAS methods, including `GrB_finalize`, will result in undefined  
1061 behavior.

### 1062 **4.1.3 getVersion: Get the version number of the standard.**

1063 Query the library for the version number of the standard that this library implements.

## 1064 **C Syntax**

```
1065         GrB_Info GrB_getVersion(unsigned int *version,  
1066                                unsigned int *subversion);
```

## 1067 **Parameters**

1068 version (OUT) On successful return will hold the value of the major version number.

1069 version (OUT) On successful return will hold the value of the subversion number.

## 1070 **Return Values**

1071 GrB\_SUCCESS operation completed successfully.

1072 GrB\_PANIC unknown internal error.

## 1073 **Description**

1074 The `getVersion` method is used to query the major and minor version number of the GraphBLAS  
1075 C API specification that the library implements at runtime. To support compile time queries the  
1076 following two macros shall also be defined by the library.

```
1077         #define GRB_VERSION      2  
1078         #define GRB_SUBVERSION  0
```

## 1079 **4.2 Object methods**

1080 This section describes methods that setup and operate on GraphBLAS opaque objects but are not  
1081 part of the the GraphBLAS math specification.

## 1082 4.2.1 Query methods

1083 The methods in this section query and, depending on the field, set internal fields of many Graph-  
1084 BLAS objects.

### 1085 4.2.1.1 get: Query the value of an object

#### 1086 C Syntax

```
1087     GrB_Info GrB_<OBJ>_get(GrB_<OBJ> o, GrB_Field field, ...);
1088
1089     GrB_Info GrB_Scalar_get(GrB_Scalar s, GrB_Field field, ...);
1090     GrB_Info GrB_Vector_get(GrB_Vector v, GrB_Field field, ...);
1091     GrB_Info GrB_Matrix_get(GrB_Matrix A, GrB_Field field, ...);
1092
1093     GrB_Info GrB_UnaryOp_get(GrB_UnaryOp op, GrB_Field field, ...);
1094     GrB_Info GrB_IndexUnaryOp_get(GrB_IndexUnaryOp op, GrB_Field field, ...);
1095     GrB_Info GrB_BinaryOp_get(GrB_BinaryOp op, GrB_Field field, ...);
1096     GrB_Info GrB_Monoid_get(GrB_Monoid op, GrB_Field field, ...);
1097     GrB_Info GrB_Semiring_get(GrB_Semiring op, GrB_Field field, ...);
1098
1099     GrB_Info GrB_Descriptor_get(GrB_Descriptor op, GrB_Field field, ...);
1100     GrB_Info GrB_Type_get(GrB_Type op, GrB_Field field, ...);
1101
1102     GrB_Info GrB_Global_get(GrB_Field field, ...);
```

#### 1103 Parameters

1104 OBJ is replaced in each signature by the object type being queried.

1105 OBJ (IN) An existing GraphBLAS object which is being queried.

1106 field (IN) The internal field being queried.

1107 ... (OUT) A pointer to a variable dependent on field to be filled with the value of the  
1108 internal field.

#### 1109 Return Value

1110 GrB\_SUCCESS The method completed successfully.

1111 GrB\_PANIC unknown internal error.

1112 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.



1113 GrB\_UNINITIALIZED\_OBJECT the desc parameter has not been initialized by a call to new.

1114 GrB\_INVALID\_VALUE invalid value set on the field, or invalid field.

## 1115 Description

1116 Queries a field of an existing GraphBLAS object. The type of ... is uniquely determined by field.

### 1117 4.2.1.2 set: Set field of an object

1118 Sets the content for a field for an existing GraphBLAS object.

## 1119 C Syntax

```
1120 GrB_Info GrB_<OBJ>_set(GrB_<OBJ> o, GrB_Field field, ...);
1121
1122 GrB_Info GrB_Scalar_set(GrB_Scalar s, GrB_Field field, ...);
1123 GrB_Info GrB_Vector_set(GrB_Vector v, GrB_Field field, ...);
1124 GrB_Info GrB_Matrix_set(GrB_Matrix A, GrB_Field field, ...);
1125
1126 GrB_Info GrB_UnaryOp_set(GrB_UnaryOp op, GrB_Field field, ...);
1127 GrB_Info GrB_IndexUnaryOp_set(GrB_IndexUnaryOp op, GrB_Field field, ...);
1128 GrB_Info GrB_BinaryOp_set(GrB_BinaryOp op, GrB_Field field, ...);
1129 GrB_Info GrB_Monoid_set(GrB_Monoid op, GrB_Field field, ...);
1130 GrB_Info GrB_Semiring_set(GrB_Semiring op, GrB_Field field, ...);
1131
1132 GrB_Info GrB_Descriptor_set(GrB_Descriptor op, GrB_Field field, ...);
1133 GrB_Info GrB_Type_set(GrB_Type op, GrB_Field field, ...);
1134
1135 GrB_Info GrB_Global_set(GrB_Field field, ...);
```

## 1136 Parameters

1137 OBJ (IN) The GraphBLAS object which is having field set.

1138 field (IN) The field being set.

1139 ... (OUT) The new value for field.

## 1140 Return Values

1141 GrB\_SUCCESS The method completed successfully.

1142 GrB\_PANIC unknown internal error.

1143        GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1144 GrB\_UNINITIALIZED\_OBJECT the desc parameter has not been initialized by a call to new.

1145        GrB\_INVALID\_VALUE invalid value set on the field, or invalid field.

1146 GrB\_OUTPUT\_NOT\_EMPTY value has already been set and may not be set again.

## 1147 Description

### 1148 4.2.2 Algebra methods

#### 1149 4.2.2.1 Type\_new: Construct a new GraphBLAS (user-defined) type

1150 Creates a new user-defined GraphBLAS type. This type can then be used to create new operators,  
 1151 monoids, semirings, vectors and matrices.

## 1152 C Syntax

```
1153         GrB_Info GrB_Type_new(GrB_Type  *utype,
1154                               size_t     sizeof(ctype));
```

## 1155 Parameters

1156        utype (INOUT) On successful return, contains a handle to the newly created user-defined  
 1157        GraphBLAS type object.

1158        ctype (IN) A C type that defines the new GraphBLAS user-defined type.

## 1159 Return Values

1160        GrB\_SUCCESS operation completed successfully.

1161        GrB\_PANIC unknown internal error.

1162        GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1163        GrB\_NULL\_POINTER utype pointer is NULL.

## 1164 Description

1165 Given a C type ctype, the Type\_new method returns in utype a handle to a new GraphBLAS type  
 1166 that is equivalent to the C type. Variables of this ctype must be a struct, union, or fixed-size array.  
 1167 In particular, given two variables, src and dst, of type ctype, the following operation must be a  
 1168 valid way to copy the contents of src to dst:

1169                   memcpy(&dst, &src, sizeof(ctype))

1170 A new, user-defined type `utype` should be destroyed with a call to `GrB_free(utype)` when no longer  
1171 needed.

1172 It is not an error to call this method more than once on the same variable; however, the handle to  
1173 the previously created object will be overwritten.

#### 1174 4.2.2.2 UnaryOp\_new: Construct a new GraphBLAS unary operator

1175 Initializes a new GraphBLAS unary operator with a specified user-defined function and its types  
1176 (domains).

#### 1177 C Syntax

```
1178     GrB_Info GrB_UnaryOp_new(GrB_UnaryOp *unary_op,  
1179                             void          (*unary_func)(void*, const void*),  
1180                             GrB_Type      d_out,  
1181                             GrB_Type      d_in);
```

#### 1182 Parameters

1183     `unary_op` (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1184     unary operator object.

1185     `unary_func` (IN) a pointer to a user-defined function that takes one input parameter of `d_in`'s  
1186     type and returns a value of `d_out`'s type, both passed as `void` pointers. Specifically  
1187     the signature of the function is expected to be of the form:

```
1188                   void func(void *out, const void *in);  
1189
```

1190     `d_out` (IN) The `GrB_Type` of the return value of the unary operator being created. Should  
1191     be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-  
1192     BLAS type.

1193     `d_in` (IN) The `GrB_Type` of the input argument of the unary operator being created.  
1194     Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1195     GraphBLAS type.

#### 1196 Return Values

1197     `GrB_SUCCESS` operation completed successfully.

1198     `GrB_PANIC` unknown internal error.



## 1228 Parameters

1229     **binary\_op** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1230     binary operator object.

1231     **binary\_func** (IN) A pointer to a user-defined function that takes two input parameters of types  
1232     **d\_in1** and **d\_in2** and returns a value of type **d\_out**, all passed as **void** pointers.  
1233     Specifically the signature of the function is expected to be of the form:

1234             **void func(void \*out, const void \*in1, const void \*in2);**  
1235

1236     **d\_out** (IN) The **GrB\_Type** of the return value of the binary operator being created. Should  
1237     be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-  
1238     BLAS type.

1239     **d\_in1** (IN) The **GrB\_Type** of the left hand argument of the binary operator being created.  
1240     Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1241     GraphBLAS type.

1242     **d\_in2** (IN) The **GrB\_Type** of the right hand argument of the binary operator being cre-  
1243     ated. Should be one of the predefined GraphBLAS types in Table 3.2, or a user-  
1244     defined GraphBLAS type.

## 1245 Return Values

1246     **GrB\_SUCCESS** operation completed successfully.

1247     **GrB\_PANIC** unknown internal error.

1248     **GrB\_OUT\_OF\_MEMORY** not enough memory available for operation.

1249     **GrB\_UNINITIALIZED\_OBJECT** the **GrB\_Type** (for user-defined types) has not been initialized by a  
1250     call to **GrB\_Type\_new**.

1251     **GrB\_NULL\_POINTER** **binary\_op** or **binary\_func** pointer is **NULL**.

## 1252 Description

1253     The **BinaryOp\_new** methods creates a new GraphBLAS binary operator

1254              $f_b = \langle \mathbf{D}(\mathbf{d\_out}), \mathbf{D}(\mathbf{d\_in1}), \mathbf{D}(\mathbf{d\_in2}), \mathbf{binary\_func} \rangle$

1255     and returns a handle to it in **binary\_op**.

1256     The implementation of **binary\_func** must be such that it works even if any of the **d\_out**, **d\_in1**, and  
1257     **d\_in2** arguments are aliased to each other. In other words, for all invocations of the function:

```
1258     binary_func(out,in1,in2);
```

1259 the value of out must be the same as if the following code was executed:

```
1260     D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));
1261     D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));
1262     memcpy(tmp1,in1,sizeof(D(d_in1)));
1263     memcpy(tmp2,in2,sizeof(D(d_in2)));
1264     binary_func(out,tmp1,tmp2);
1265     free(tmp2);
1266     free(tmp1);
```

1267 It is not an error to call this method more than once on the same variable; however, the handle to  
1268 the previously created object will be overwritten.

#### 1269 4.2.2.4 Monoid\_new: Construct a new GraphBLAS monoid

1270 Creates a new monoid with specified binary operator and identity value.

#### 1271 C Syntax

```
1272     GrB_Info GrB_Monoid_new(GrB_Monoid    *monoid,
1273                             GrB_BinaryOp   binary_op,
1274                             <type>        identity);
```

#### 1275 Parameters

1276 monoid (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1277 monoid object.

1278 binary\_op (IN) An existing GraphBLAS associative binary operator whose input and output  
1279 types are the same.

1280 identity (IN) The value of the identity element of the monoid. Must be the same type as  
1281 the type used by the binary\_op operator.

#### 1282 Return Values

1283 GrB\_SUCCESS operation completed successfully.

1284 GrB\_PANIC unknown internal error.

1285 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1286 GrB\_UNINITIALIZED\_OBJECT the GrB\_BinaryOp (for user-defined operators) has not been initial-  
1287 ized by a call to GrB\_BinaryOp\_new.

1288 GrB\_NULL\_POINTER monoid pointer is NULL.

1289 GrB\_DOMAIN\_MISMATCH all three argument types of the binary operator and the type of the  
1290 identity value are not the same.

## 1291 Description

1292 The Monoid\_new method creates a new monoid  $M = \langle \mathbf{D}(\text{binary\_op}), \text{binary\_op}, \text{identity} \rangle$  and re-  
1293 turns a handle to it in monoid.

1294 If binary\_op is not associative, the results of GraphBLAS operations that require associativity of  
1295 this monoid will be undefined.

1296 It is not an error to call this method more than once on the same variable; however, the handle to  
1297 the previously created object will be overwritten.

### 1298 4.2.2.5 Semiring\_new: Construct a new GraphBLAS semiring

1299 Creates a new semiring with specified domain, operators, and elements.

## 1300 C Syntax

```
1301 GrB_Info GrB_Semiring_new(GrB_Semiring *semiring,  
1302                           GrB_Monoid   add_op,  
1303                           GrB_BinaryOp  mul_op);
```

## 1304 Parameters

1305 semiring (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1306 semiring.

1307 add\_op (IN) An existing GraphBLAS commutative monoid that specifies the addition op-  
1308 erator and its identity.

1309 mul\_op (IN) An existing GraphBLAS binary operator that specifies the semiring's multi-  
1310 plication operator. In addition, mul\_op's output domain,  $\mathbf{D}_{out}(\text{mul\_op})$ , must be  
1311 the same as the add\_op's domain  $\mathbf{D}(\text{add\_op})$ .

## 1312 Return Values

1313 GrB\_SUCCESS operation completed successfully.

1314                   GrB\_PANIC unknown internal error.

1315           GrB\_OUT\_OF\_MEMORY not enough memory available for this method to complete.

1316 GrB\_UNINITIALIZED\_OBJECT the `add_op` (for user-define monoids) object has not been initialized  
1317                               with a call to `GrB_Monoid_new` or the `mul_op` (for user-defined  
1318                               operators) object has not been not been initialized by a call to  
1319                               `GrB_BinaryOp_new`.

1320           GrB\_NULL\_POINTER semiring pointer is NULL.

1321           GrB\_DOMAIN\_MISMATCH the output domain of `mul_op` does not match the domain of the  
1322                               `add_op` monoid.

## 1323 Description

1324 The `Semiring_new` method creates a new semiring:

$$1325 \quad S = \langle \mathbf{D}_{out}(\text{mul\_op}), \mathbf{D}_{in_1}(\text{mul\_op}), \mathbf{D}_{in_2}(\text{mul\_op}), \text{add\_op}, \text{mul\_op}, \mathbf{0}(\text{add\_op}) \rangle$$

1326 and returns a handle to it in `semiring`. Note that  $\mathbf{D}_{out}(\text{mul\_op})$  must be the same as  $\mathbf{D}(\text{add\_op})$ .  
1327 If `add_op` is not commutative, then GraphBLAS operations using this semiring will be undefined.  
1328 It is not an error to call this method more than once on the same variable; however, the handle to  
1329 the previously created object will be overwritten.

### 1330 4.2.2.6 IndexUnaryOp\_new: Construct a new GraphBLAS index unary operator [Scott: 1331 NEW CONTENT]

1332 Initializes a new GraphBLAS index unary operator with a specified user-defined function and its  
1333 types (domains).

## 1334 C Syntax

```

1335 GrB_Info GrB_IndexUnaryOp_new(GrB_IndexUnaryOp  *index_unary_op,
1336                               void (*index_unary_func)(void*,
1337                                                         const void*,
1338                                                         GrB_Index,
1339                                                         GrB_Index,
1340                                                         const void*),
1341                               GrB_Type          d_out,
1342                               GrB_Type          d_in1,
1343                               GrB_Type          d_in2);

```



## 1344 Parameters

1345 **index\_unary\_op** (INOUT) On successful return, contains a handle to the newly created Graph-  
1346 BLAS index unary operator object.

1347 **index\_unary\_func** (IN) A pointer to a user-defined function that takes input parameters of types  
1348 **d\_in1**, **GrB\_Index**, **GrB\_Index** and **d\_in2** and returns a value of type **d\_out**. Ex-  
1349 cept for the **GrB\_Index** parameters, all are passed as **void** pointers. Specifically  
1350 the signature of the function is expected to be of the form:

```
1351         void func(void      *out,  
1352                   const void *in1,  
1353                   GrB_Index  row_index,  
1354                   GrB_Index  col_index,  
1355                   const void *in2);
```

1357 **d\_out** (IN) The **GrB\_Type** of the return value of the index unary operator being created.  
1358 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1359 GraphBLAS type.

1360 **d\_in1** (IN) The **GrB\_Type** of the first input argument of the index unary operator being  
1361 created and corresponds to the stored values of the **GrB\_Vector** or **GrB\_Matrix**  
1362 being operated on. Should be one of the predefined GraphBLAS types in Ta-  
1363 ble 3.2, or a user-defined GraphBLAS type.

1364 **d\_in2** (IN) The **GrB\_Type** of the last input argument of the index unary operator be-  
1365 ing created and corresponds to a scalar provided by the GraphBLAS operation  
1366 that uses this operator. Should be one of the predefined GraphBLAS types in  
1367 Table 3.2, or a user-defined GraphBLAS type.

## 1368 Return Values

1369 **GrB\_SUCCESS** operation completed successfully.

1370 **GrB\_PANIC** unknown internal error.

1371 **GrB\_OUT\_OF\_MEMORY** not enough memory available for operation.

1372 **GrB\_UNINITIALIZED\_OBJECT** the **GrB\_Type** (for user-defined types) has not been initialized by a  
1373 call to **GrB\_Type\_new**.

1374 **GrB\_NULL\_POINTER** **index\_unary\_op** or **index\_unary\_func** pointer is **NULL**.

## 1375 Description

1376 The **IndexUnaryOp\_new** methods creates a new GraphBLAS index unary operator

1377  $f_i = \langle \mathbf{D}(d\_out), \mathbf{D}(d\_in1), \mathbf{D}(GrB\_Index), \mathbf{D}(GrB\_Index), \mathbf{D}(d\_in2), index\_unary\_func) \rangle$

1378 and returns a handle to it in `index_unary_op`.

1379 The implementation of `index_unary_func` must be such that it works even if any of the `d_out`,  
 1380 `d_in1`, and `d_in2` arguments are aliased to each other. In other words, for all invocations of the  
 1381 function:

1382 `index_unary_func(out,in1,row_index,col_index,n,in2);`

1383 the value of `out` must be the same as if the following code was executed (shown here for matrices):

```
1384 GrB_Index row_index = ...;
1385 GrB_Index col_index = ...;
1386 D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));
1387 D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));
1388 memcpy(tmp1,in1,sizeof(D(d_in1)));
1389 memcpy(tmp2,in2,sizeof(D(d_in2)));
1390 index_unary_func(out,tmp1,row_index,col_index,tmp2);
1391 free(tmp2);
1392 free(tmp1);
```

1393 It is not an error to call this method more than once on the same variable; however, the handle to  
 1394 the previously created object will be overwritten.

## 1395 4.2.3 Scalar methods

### 1396 4.2.3.1 Scalar\_new: Construct a new scalar

1397 Creates a new empty scalar with specified domain.

## 1398 C Syntax

```
1399 GrB_Info GrB_Scalar_new(GrB_Scalar *s,
1400                        GrB_Type d);
```

## 1401 Parameters

1402 **s** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
 1403 scalar.

1404 **d** (IN) The type corresponding to the domain of the scalar being created. Can be  
 1405 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
 1406 GraphBLAS type.

## 1407 Return Values

1408           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1409                       blocking mode, this indicates that the API checks for the input  
1410                       arguments passed successfully. Either way, output scalar `s` is ready  
1411                       to be used in the next method of the sequence.

1412           GrB\_PANIC Unknown internal error.

1413           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1414                       GraphBLAS objects (input or output) is in an invalid state caused  
1415                       by a previous execution error. Call `GrB_error()` to access any error  
1416                       messages generated by the implementation.

1417           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1418 GrB\_UNINITIALIZED\_OBJECT The `GrB_Type` object has not been initialized by a call to `GrB_Type_new`  
1419                       (needed for user-defined types).

1420           GrB\_NULL\_POINTER The `s` pointer is NULL.

## 1421 Description

1422 Creates a new GraphBLAS scalar `s` of domain `D(d)` and empty `L(s)`. The method returns a handle  
1423 to the new scalar in `s`.

1424 It is not an error to call this method more than once on the same variable; however, the handle to  
1425 the previously created object will be overwritten.

### 1426 4.2.3.2 Scalar\_dup: Construct a copy of a GraphBLAS scalar

1427 Creates a new scalar with the same domain and contents as another scalar.

## 1428 C Syntax

```
1429           GrB_Info GrB_Scalar_dup(GrB_Scalar        *t,  
1430                                    const GrB_Scalar  s);
```

## 1431 Parameters

1432           t (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1433           scalar.

1434           s (IN) The GraphBLAS scalar to be duplicated.

## 1435 Return Values

1436           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1437                       blocking mode, this indicates that the API checks for the input  
1438                       arguments passed successfully. Either way, output scalar *t* is ready  
1439                       to be used in the next method of the sequence.

1440           GrB\_PANIC Unknown internal error.

1441           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1442                       GraphBLAS objects (input or output) is in an invalid state caused  
1443                       by a previous execution error. Call `GrB_error()` to access any error  
1444                       messages generated by the implementation.

1445           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1446 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS scalar, *s*, has not been initialized by a call to  
1447                       `Scalar_new` or `Scalar_dup`.

1448           GrB\_NULL\_POINTER The *t* pointer is NULL.

## 1449 Description

1450 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  
1451 scalar in *t*.

1452 It is not an error to call this method more than once with the same output variable; however, the  
1453 handle to the previously created object will be overwritten.

### 1454 4.2.3.3 Scalar\_clear: Clear/remove a stored value from a scalar

1455 Removes the stored value from a scalar.

## 1456 C Syntax

1457           GrB\_Info GrB\_Scalar\_clear(GrB\_Scalar s);

## 1458 Parameters

1459           *s* (INOUT) An existing GraphBLAS scalar to clear.

## 1460 Return Values

1461           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1462                       blocking mode, this indicates that the API checks for the input

1463 arguments passed successfully. Either way, output scalar `s` is ready  
 1464 to be used in the next method of the sequence.

1465 `GrB_PANIC` Unknown internal error.

1466 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque  
 1467 GraphBLAS objects (input or output) is in an invalid state caused  
 1468 by a previous execution error. Call `GrB_error()` to access any error  
 1469 messages generated by the implementation.

1470 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1471 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS scalar, `s`, has not been initialized by a call to  
 1472 `Scalar_new` or `Scalar_dup`.

## 1473 Description

1474 Removes the stored value from an existing scalar. After the call, `L(s)` is empty. The size of the  
 1475 scalar does not change.

### 1476 4.2.3.4 `Scalar_nvals`: Number of stored elements in a scalar

1477 Retrieve the number of stored elements in a scalar (either zero or one).

## 1478 C Syntax

```
1479      GrB_Info GrB_Scalar_nvals(GrB_Index      *nvals,
1480                               const GrB_Scalar s);
```

## 1481 Parameters

1482 `nvals` (OUT) On successful return, this is set to the number of stored elements in the  
 1483 scalar (zero or one).

1484 `s` (IN) An existing GraphBLAS scalar being queried.

## 1485 Return Values

1486 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-  
 1487 cessfully and the value of `nvals` has been set.

1488 `GrB_PANIC` Unknown internal error.

1489       GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 1490       GraphBLAS objects (input or output) is in an invalid state caused  
 1491       by a previous execution error. Call GrB\_error() to access any error  
 1492       messages generated by the implementation.

1493       GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1494   GrB\_UNINITIALIZED\_OBJECT The GraphBLAS scalar, *s*, has not been initialized by a call to  
 1495       Scalar\_new or Scalar\_dup.

1496       GrB\_NULL\_POINTER The *nvals* pointer is NULL.

## 1497 Description

1498 Return *nvals*(*s*) in *nvals*. This is the number of stored elements in scalar *s*, which is the size of  
 1499 *L*(*s*), and can only be either zero or one (see Section 3.5.1).

### 1500 4.2.3.5 Scalar\_setElement: Set the single element in a scalar

1501 Set the single element of a scalar to a given value.

## 1502 C Syntax

```
1503       GrB_Info GrB_Scalar_setElement(GrB_Scalar    s,  

1504                                                    <type>       val);
```

## 1505 Parameters

1506       *s* (INOUT) An existing GraphBLAS scalar for which the element is to be assigned.

1507       *val* (IN) Scalar value to assign. The type must be compatible with the domain of *s*.

## 1508 Return Values

1509       GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 1510       blocking mode, this indicates that the compatibility tests on in-  
 1511       dex/dimensions and domains for the input arguments passed suc-  
 1512       cessfully. Either way, the output scalar *s* is ready to be used in the  
 1513       next method of the sequence.

1514       GrB\_PANIC Unknown internal error.

1515       GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 1516       GraphBLAS objects (input or output) is in an invalid state caused

1517 by a previous execution error. Call `GrB_error()` to access any error  
1518 messages generated by the implementation.

1519 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1520 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS scalar, `s`, has not been initialized by a call to  
1521 `Scalar_new` or `Scalar_dup`.

1522 `GrB_DOMAIN_MISMATCH` The domains of `s` and `val` are incompatible.

## 1523 Description

1524 First, `val` and output GraphBLAS scalar are tested for domain compatibility as follows: `D(val)` must  
1525 be compatible with `D(s)`. Two domains are compatible with each other if values from one domain  
1526 can be cast to values in the other domain as per the rules of the C language. In particular, domains  
1527 from Table 3.2 are all compatible with each other. A domain from a user-defined type is only com-  
1528 patible with itself. If any compatibility rule above is violated, execution of `GrB_Scalar_setElement`  
1529 ends and the domain mismatch error listed above is returned.

1530 We are now ready to carry out the assignment `val`; that is:

$$1531 \quad s(0) = val$$

1532 If `s` already had a stored value, it will be overwritten; otherwise, the new value is stored in `s`.

1533 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents  
1534 of `s` is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with  
1535 return value `GrB_SUCCESS` and the new content of scalar `s` is as defined above but may not be  
1536 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 1537 4.2.3.6 `Scalar_extractElement`: Extract a single element from a scalar.

1538 Assign a non-opaque scalar with the value of the element stored in a GraphBLAS scalar.

## 1539 C Syntax

```
1540     GrB_Info GrB_Scalar_extractElement(<type>          *val,  
1541                                     const GrB_Scalar s);
```

## 1542 Parameters

1543 `val` (INOUT) Pointer to a non-opaque scalar of type that is compatible with the domain  
1544 of scalar `s`. On successful return, `val` holds the result of the operation, and any  
1545 previous value in `val` is overwritten.

1546 `s` (IN) The GraphBLAS scalar from which an element is extracted.





## 1577 4.2.4 Vector methods

### 1578 4.2.4.1 Vector\_new: Construct new vector

1579 Creates a new vector with specified domain and size.

## 1580 C Syntax

```
1581         GrB_Info GrB_Vector_new(GrB_Vector *v,  
1582                                GrB_Type    d,  
1583                                GrB_Index   nsize);
```

## 1584 Parameters

1585 v (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1586 vector.

1587 d (IN) The type corresponding to the domain of the vector being created. Can be  
1588 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
1589 GraphBLAS type.

1590 nsize (IN) The size of the vector being created.

## 1591 Return Values

1592 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1593 blocking mode, this indicates that the API checks for the input  
1594 arguments passed successfully. Either way, output vector v is ready  
1595 to be used in the next method of the sequence.

1596 GrB\_PANIC Unknown internal error.

1597 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1598 GraphBLAS objects (input or output) is in an invalid state caused  
1599 by a previous execution error. Call GrB\_error() to access any error  
1600 messages generated by the implementation.

1601 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1602 GrB\_UNINITIALIZED\_OBJECT The GrB\_Type object has not been initialized by a call to GrB\_Type\_new  
1603 (needed for user-defined types).

1604 GrB\_NULL\_POINTER The v pointer is NULL.

1605 GrB\_INVALID\_VALUE nsize is zero or outside the range of the type GrB\_Index.

## 1606 Description

1607 Creates a new vector  $\mathbf{v}$  of domain  $\mathbf{D(d)}$ , size  $\mathbf{nsz}$ , and empty  $\mathbf{L(v)}$ . The method returns a handle  
1608 to the new vector in  $\mathbf{v}$ .

1609 It is not an error to call this method more than once on the same variable; however, the handle to  
1610 the previously created object will be overwritten.

### 1611 4.2.4.2 Vector\_dup: Construct a copy of a GraphBLAS vector

1612 Creates a new vector with the same domain, size, and contents as another vector.

## 1613 C Syntax

```
1614         GrB_Info GrB_Vector_dup(GrB_Vector      *w,  
1615                                const GrB_Vector  u);
```

## 1616 Parameters

1617  $\mathbf{w}$  (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1618 vector.

1619  $\mathbf{u}$  (IN) The GraphBLAS vector to be duplicated.

## 1620 Return Values

1621  $\mathbf{GrB\_SUCCESS}$  In blocking mode, the operation completed successfully. In non-  
1622 blocking mode, this indicates that the API checks for the input  
1623 arguments passed successfully. Either way, output vector  $\mathbf{w}$  is ready  
1624 to be used in the next method of the sequence.

1625  $\mathbf{GrB\_PANIC}$  Unknown internal error.

1626  $\mathbf{GrB\_INVALID\_OBJECT}$  This is returned in any execution mode whenever one of the opaque  
1627 GraphBLAS objects (input or output) is in an invalid state caused  
1628 by a previous execution error. Call  $\mathbf{GrB\_error()}$  to access any error  
1629 messages generated by the implementation.

1630  $\mathbf{GrB\_OUT\_OF\_MEMORY}$  Not enough memory available for operation.

1631  $\mathbf{GrB\_UNINITIALIZED\_OBJECT}$  The GraphBLAS vector,  $\mathbf{u}$ , has not been initialized by a call to  
1632  $\mathbf{Vector\_new}$  or  $\mathbf{Vector\_dup}$ .

1633  $\mathbf{GrB\_NULL\_POINTER}$  The  $\mathbf{w}$  pointer is  $\mathbf{NULL}$ .

## 1634 Description

1635 Creates a new vector **w** of domain **D(u)**, size **size(u)**, and contents **L(u)**. The method returns a  
1636 handle to the new vector in **w**.

1637 It is not an error to call this method more than once on the same variable; however, the handle to  
1638 the previously created object will be overwritten.

### 1639 4.2.4.3 Vector\_resize: Resize a vector

1640 Changes the size of an existing vector.

## 1641 C Syntax

```
1642         GrB_Info GrB_Vector_resize(GrB_Vector  w,  
1643                                   GrB_Index   nsize);
```

## 1644 Parameters

1645 w (INOUT) An existing Vector object that is being resized.

1646 nsize (IN) The new size of the vector. It can be smaller or larger than the current size.

## 1647 Return Values

1648 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1649 blocking mode, this indicates that the API checks for the input  
1650 arguments passed successfully. Either way, output vector **w** is ready  
1651 to be used in the next method of the sequence.

1652 GrB\_PANIC Unknown internal error.

1653 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1654 GraphBLAS objects (input or output) is in an invalid state caused  
1655 by a previous execution error. Call **GrB\_error()** to access any error  
1656 messages generated by the implementation.

1657 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1658 GrB\_NULL\_POINTER The **w** pointer is **NULL**.

1659 GrB\_INVALID\_VALUE **nsize** is zero or outside the range of the type **GrB\_Index**.

## 1660 Description

1661 Changes the size of  $w$  to  $nsz$ . The domain  $\mathbf{D}(w)$  of vector  $w$  remains the same. The contents  $\mathbf{L}(w)$   
1662 are modified as described below.

1663 Let  $w = \langle \mathbf{D}(w), N, \mathbf{L}(w) \rangle$  when the method is called. When the method returns,  $w = \langle \mathbf{D}(w), nsz, \mathbf{L}'(w) \rangle$   
1664 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  
1665 than or equal to the new vector size ( $nsz$ ) are dropped.

### 1666 4.2.4.4 Vector\_clear: Clear a vector

1667 Removes all the elements (tuples) from a vector.

## 1668 C Syntax

```
1669 GrB_Info GrB_Vector_clear(GrB_Vector v);
```

## 1670 Parameters

1671  $v$  (INOUT) An existing GraphBLAS vector to clear.

## 1672 Return Values

1673 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1674 blocking mode, this indicates that the API checks for the input  
1675 arguments passed successfully. Either way, output vector  $v$  is ready  
1676 to be used in the next method of the sequence.

1677 **GrB\_PANIC** Unknown internal error.

1678 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1679 GraphBLAS objects (input or output) is in an invalid state caused  
1680 by a previous execution error. Call `GrB_error()` to access any error  
1681 messages generated by the implementation.

1682 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1683 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector,  $v$ , has not been initialized by a call to  
1684 `Vector_new` or `Vector_dup`.

## 1685 Description

1686 Removes all elements (tuples) from an existing vector. After the call to `GrB_Vector_clear(v)`,  
1687  $\mathbf{L}(v) = \emptyset$ . The size of the vector does not change.

#### 1688 4.2.4.5 Vector\_size: Size of a vector

1689 Retrieve the size of a vector.

#### 1690 C Syntax

```
1691         GrB_Info GrB_Vector_size(GrB_Index          *nsize,  
1692                                const GrB_Vector  v);
```

#### 1693 Parameters

1694 nsize (OUT) On successful return, is set to the size of the vector.

1695 v (IN) An existing GraphBLAS vector being queried.

#### 1696 Return Values

1697 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
1698 cessfully and the value of nsize has been set.

1699 GrB\_PANIC Unknown internal error.

1700 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1701 GraphBLAS objects (input or output) is in an invalid state caused  
1702 by a previous execution error. Call GrB\_error() to access any error  
1703 messages generated by the implementation.

1704 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, v, has not been initialized by a call to  
1705 Vector\_new or Vector\_dup.

1706 GrB\_NULL\_POINTER nsize pointer is NULL.

#### 1707 Description

1708 Return **size**(v) in nsize.

#### 1709 4.2.4.6 Vector\_nvals: Number of stored elements in a vector

1710 Retrieve the number of stored elements (tuples) in a vector.

#### 1711 C Syntax

```
1712         GrB_Info GrB_Vector_nvals(GrB_Index          *nvals,  
1713                                const GrB_Vector  v);
```

## 1714 Parameters

1715            **nvals** (OUT) On successful return, this is set to the number of stored elements (tuples)  
1716            in the vector.

1717            **v** (IN) An existing GraphBLAS vector being queried.

## 1718 Return Values

1719            **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
1720            cessfully and the value of **nvals** has been set.

1721            **GrB\_PANIC** Unknown internal error.

1722            **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1723            GraphBLAS objects (input or output) is in an invalid state caused  
1724            by a previous execution error. Call **GrB\_error()** to access any error  
1725            messages generated by the implementation.

1726            **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1727            **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **v**, has not been initialized by a call to  
1728            **Vector\_new** or **Vector\_dup**.

1729            **GrB\_NULL\_POINTER** The **nvals** pointer is **NULL**.

## 1730 Description

1731            Return **nvals(v)** in **nvals**. This is the number of stored elements in vector **v**, which is the size of  
1732            **L(v)** (see Section 3.5.2).

### 1733 4.2.4.7 Vector\_build: Store elements from tuples into a vector

## 1734 C Syntax

```
1735            GrB_Info GrB_Vector_build(GrB_Vector            w,  
1736                                        const GrB_Index        *indices,  
1737                                        const <type>           *values,  
1738                                        GrB_Index                n,  
1739                                        const GrB_BinaryOp       dup);
```

## 1740 Parameters

1741            **w** (INOUT) An existing Vector object to store the result.

1742            **indices** (IN) Pointer to an array of indices.

1743        **values** (IN) Pointer to an array of scalars of a type that is compatible with the domain of  
 1744        vector **w**.

1745        **n** (IN) The number of entries contained in each array (the same for indices and values).

1746        **dup** (IN) An associative and commutative binary operator to apply when duplicate  
 1747        values for the same location are present in the input arrays. All three domains of  
 1748        **dup** must be the same; hence  $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$ . If **dup** is **GrB\_NULL**,  
 1749        then duplicate locations will result in an error.

## 1750 Return Values

1751        **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 1752        blocking mode, this indicates that the API checks for the input  
 1753        arguments passed successfully. Either way, output vector **w** is  
 1754        ready to be used in the next method of the sequence.

1755        **GrB\_PANIC** Unknown internal error.

1756        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
 1757        opaque GraphBLAS objects (input or output) is in an invalid  
 1758        state caused by a previous execution error. Call **GrB\_error()** to  
 1759        access any error messages generated by the implementation.

1760        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1761        **GrB\_UNINITIALIZED\_OBJECT** Either **w** has not been initialized by a call to **GrB\_Vector\_new**  
 1762        or by **GrB\_Vector\_dup**, or **dup** has not been initialized by a call  
 1763        to **GrB\_BinaryOp\_new**.

1764        **GrB\_NULL\_POINTER** indices or values pointer is **NULL**.

1765        **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in indices is outside the allowed range for **w**.

1766        **GrB\_DOMAIN\_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are  
 1767        not all the same, or the domains of **values** and **w** are incompatible  
 1768        with each other or  $D_{dup}$ .

1769        **GrB\_OUTPUT\_NOT\_EMPTY** Output vector **w** already contains valid tuples (elements). In  
 1770        other words, **GrB\_Vector\_nvals(C)** returns a positive value.

1771        **GrB\_INVALID\_VALUE** indices contains a duplicate location and **dup** is **GrB\_NULL**.

## 1772 Description

1773        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  
 1774        from **w** in its domain; otherwise,  $\tilde{\mathbf{w}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \emptyset \rangle$ .

1775 Each tuple  $\{\text{indices}[k], \text{values}[k]\}$ , where  $0 \leq k < n$ , is a contribution to the output in the form of

$$1776 \quad \tilde{\mathbf{w}}(\text{indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } dup \neq \text{GrB\_NULL} \\ (\mathbf{D}(\mathbf{w})) \text{values}[k] & \text{otherwise.} \end{cases}$$

1777 If multiple values for the same location are present in the input arrays and `dup` is not `GrB_NULL`,  
1778 `dup` is used to reduce the values before assignment into  $\tilde{\mathbf{w}}$  as follows:

$$1779 \quad \tilde{\mathbf{w}}_i = \bigoplus_{k: \text{indices}[k]=i} (D_{dup}) \text{values}[k],$$

1780 where  $\oplus$  is the `dup` binary operator. Finally, the resulting  $\tilde{\mathbf{w}}$  is copied into  $\mathbf{w}$  via typecasting its  
1781 values to  $\mathbf{D}(\mathbf{w})$  if necessary. If  $\oplus$  is not associative or not commutative, the result is undefined.

1782 The nonopaque input arrays, `indices` and `values`, must be at least as large as `n`.

1783 It is an error to call this function on an output object with existing elements. In other words,  
1784 `GrB_Vector_nvals(w)` should evaluate to zero prior to calling this function.

1785 After `GrB_Vector_build` returns, it is safe for a programmer to modify or delete the arrays `indices`  
1786 or `values`.

#### 1787 4.2.4.8 Vector\_setElement: Set a single element in a vector

1788 Set one element of a vector to a given value.

#### 1789 C Syntax

```
1790 // scalar value
1791 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1792                               <type>         val,
1793                               GrB_Index       index);
1794
1795 // GraphBLAS scalar
1796 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1797                               const GrB_Scalar s,
1798                               GrB_Index       index);
```

#### 1799 Parameters

1800 `w` (INOUT) An existing GraphBLAS vector for which an element is to be assigned.

1801 `val` or `s` (IN) Scalar assign. Its domain (type) must be compatible with the domain of `w`.

1802 `index` (IN) The location of the element to be assigned.



## 1803 Return Values

1804           **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1805                           blocking mode, this indicates that the compatibility tests on in-  
1806                           dex/dimensions and domains for the input arguments passed suc-  
1807                           cessfully. Either way, the output vector **w** is ready to be used in  
1808                           the next method of the sequence.

1809           **GrB\_PANIC** Unknown internal error.

1810           **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1811                           GraphBLAS objects (input or output) is in an invalid state caused  
1812                           by a previous execution error. Call **GrB\_error()** to access any error  
1813                           messages generated by the implementation.

1814           **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1815 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **w**, or GraphBLAS scalar, **s**, has not been  
1816                           initialized by a call to a respective constructor.

1817           **GrB\_INVALID\_INDEX** **index** specifies a location that is outside the dimensions of **w**.

1818           **GrB\_DOMAIN\_MISMATCH** The domains of the vector and the scalar are incompatible.

## 1819 Description

1820 First, the scalar and output vector are tested for domain compatibility as follows: **D(val)** or **D(s)**  
1821 must be compatible with **D(w)**. Two domains are compatible with each other if values from  
1822 one domain can be cast to values in the other domain as per the rules of the C language. In  
1823 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
1824 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
1825 **GrB\_Vector\_setElement** ends and the domain mismatch error listed above is returned.

1826 Then, the **index** parameter is checked for a valid value where the following condition must hold:

$$1827 \qquad 0 \leq \text{index} < \text{size}(\mathbf{w})$$

1828 If this condition is violated, execution of **GrB\_Vector\_setElement** ends and the invalid index error  
1829 listed above is returned.

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

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

1830 In the case of a transparent scalar or if **L(s)** is not empty, then a value will be stored at the  
1831 specified location in **w**, overwriting any value that may have been stored there before. In the case  
1832 of a GraphBLAS scalar, if **L(s)** is empty, then any value stored at the specified location in **w** will  
1833 be removed.

1834 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new contents  
 1835 of **w** is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method exits with  
 1836 return value GrB\_SUCCESS and the new contents of vector **w** is as defined above but may not be  
 1837 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 1838 4.2.4.9 Vector\_removeElement: Remove an element from a vector

1839 Remove (annihilate) one stored element from a vector.

### 1840 C Syntax

```
1841      GrB_Info GrB_Vector_removeElement(GrB_Vector  w,
1842                                     GrB_Index    index);
```

### 1843 Parameters

1844 **w** (INOUT) An existing GraphBLAS vector from which an element is to be removed.

1845 **index** (IN) The location of the element to be removed.

### 1846 Return Values

1847 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 1848 blocking mode, this indicates that the compatibility tests on in-  
 1849 dex/dimensions and domains for the input arguments passed suc-  
 1850 cessfully. Either way, the output vector **w** is ready to be used in  
 1851 the next method of the sequence.

1852 **GrB\_PANIC** Unknown internal error.

1853 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 1854 GraphBLAS objects (input or output) is in an invalid state caused  
 1855 by a previous execution error. Call **GrB\_error()** to access any error  
 1856 messages generated by the implementation.

1857 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1858 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **w**, has not been initialized by a call to  
 1859 **Vector\_new** or **Vector\_dup**.

1860 **GrB\_INVALID\_INDEX** **index** specifies a location that is outside the dimensions of **w**.

## 1861 Description

1862 First, the `index` parameter is checked for a valid value where the following condition must hold:

$$1863 \quad 0 \leq \text{index} < \text{size}(\mathbf{w})$$

1864 If this condition is violated, execution of `GrB_Vector_removeElement` ends and the invalid index  
1865 error listed above is returned.

1866 We are now ready to carry out the removal of a value that may be stored at the location specified  
1867 by `index`. If a value does not exist at the specified location in  $\mathbf{w}$ , no error is reported and the  
1868 operation has no effect on the state of  $\mathbf{w}$ . In either case, the following will be true on return from  
1869 the method: `index`  $\notin$  `ind(w)`.

1870 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents  
1871 of  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with  
1872 return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but may not be  
1873 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 1874 4.2.4.10 Vector\_extractElement: Extract a single element from a vector.

1875 Extract one element of a vector into a scalar.

## 1876 C Syntax

```
1877 // scalar value
1878 GrB_Info GrB_Vector_extractElement(<type>          *val,
1879                                   const GrB_Vector  u,
1880                                   GrB_Index         index);
1881
1882 // GraphBLAS scalar
1883 GrB_Info GrB_Vector_extractElement(GrB_Scalar      s,
1884                                   const GrB_Vector  u,
1885                                   GrB_Index         index);
```

## 1886 Parameters

1887 `val` or `s` (INOUT) An existing scalar of whose domain is compatible with the domain of vector  
1888 `u`. On successful return, this scalar holds the result of the extract. Any previous  
1889 value stored in `val` or `s` is overwritten.

1890 `u` (IN) The GraphBLAS vector from which an element is extracted.

1891 `index` (IN) The location in `u` to extract.

## 1892 Return Values

1893           GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
 1894                           cessfully. This indicates that the compatibility tests on dimensions  
 1895                           and domains for the input arguments passed successfully, and the  
 1896                           output scalar, **val** or **s**, has been computed and is ready to be used  
 1897                           in the next method of the sequence.

1898           GrB\_NO\_VALUE When using the transparent scalar, **val**, this is returned when there  
 1899                           is no stored value at specified location.

1900           GrB\_PANIC Unknown internal error.

1901           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 1902                           GraphBLAS objects (input or output) is in an invalid state caused  
 1903                           by a previous execution error. Call **GrB\_error()** to access any error  
 1904                           messages generated by the implementation.

1905           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1906 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, **u**, or scalar, **s**, has not been initialized by  
 1907                           a call to a corresponding constructor.

1908           GrB\_NULL\_POINTER **val** pointer is NULL.

1909           GrB\_INVALID\_INDEX **index** specifies a location that is outside the dimensions of **w**.

1910           GrB\_DOMAIN\_MISMATCH The domains of the vector and scalar are incompatible.

## 1911 Description

1912 First, the scalar and input vector are tested for domain compatibility as follows: **D(val)** or **D(s)**  
 1913 must be compatible with **D(u)**. Two domains are compatible with each other if values from  
 1914 one domain can be cast to values in the other domain as per the rules of the C language. In  
 1915 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
 1916 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
 1917 **GrB\_Vector\_extractElement** ends and the domain mismatch error listed above is returned.

1918 Then, the **index** parameter is checked for a valid value where the following condition must hold:

$$1919 \qquad 0 \leq \text{index} < \text{size}(\mathbf{u})$$

1920 If this condition is violated, execution of **GrB\_Vector\_extractElement** ends and the invalid index  
 1921 error listed above is returned.

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

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

1922 If  $\text{index} \in \mathbf{ind}(u)$ , then the corresponding value from  $u$  is copied into  $s$  or  $val$  with casting as  
 1923 necessary. If  $\text{index} \notin \mathbf{ind}(u)$ , then one of the follow occurs depending on output scalar type:

- 1924 • The GraphBLAS scalar,  $s$ , is cleared and `GrB_SUCCESS` is returned.
- 1925 • The non-opaque scalar,  $val$ , is unchanged, and `GrB_NO_VALUE` is returned.

1926 When using the non-opaque scalar variant ( $val$ ) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`  
 1927 mode, the new contents of  $val$  are as defined above if the method exits with return value `GrB_SUCCESS`  
 1928 or `GrB_NO_VALUE`.

1929 When using the GraphBLAS scalar variant ( $s$ ) with a `GrB_SUCCESS` return value, the method  
 1930 exits and the new contents of  $s$  is as defined above and fully computed in `GrB_BLOCKING` mode.  
 1931 In `GrB_NONBLOCKING` mode, the new contents of  $s$  is as defined above but may not be fully  
 1932 computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 1933 4.2.4.11 Vector\_extractTuples: Extract tuples from a vector

1934 Extract the contents of a GraphBLAS vector into non-opaque data structures.

#### 1935 C Syntax

```
1936      GrB_Info GrB_Vector_extractTuples(GrB_Index      *indices,
1937                                     <type>          *values,
1938                                     GrB_Index        *n,
1939                                     const GrB_Vector  v);
1940
```

1941 **indices** (OUT) Pointer to an array of indices that is large enough to hold all of the stored  
 1942 values' indices.

1943 **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of  
 1944 the stored values whose type is compatible with  $\mathbf{D}(v)$ .

1945 **n** (INOUT) Pointer to a value indicating (on input) the number of elements the  
 1946 values and indices arrays can hold. Upon return, it will contain the number of  
 1947 values written to the arrays.

1948 **v** (IN) An existing GraphBLAS vector.

#### 1949 Return Values

1950 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
 1951 cessfully. This indicates that the compatibility tests on the input  
 1952 argument passed successfully, and the output arrays, **indices** and  
 1953 **values**, have been computed.

1954                   GrB\_PANIC Unknown internal error.

1955           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1956                   GraphBLAS objects (input or output) is in an invalid state caused  
1957                   by a previous execution error. Call GrB\_error() to access any error  
1958                   messages generated by the implementation.

1959           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1960           GrB\_INSUFFICIENT\_SPACE Not enough space in `indices` and `values` (as indicated by the `n` pa-  
1961                   rameter) to hold all of the tuples that will be extracted.

1962 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, `v`, has not been initialized by a call to  
1963                   Vector\_new or Vector\_dup.

1964           GrB\_NULL\_POINTER `indices`, `values`, or `n` pointer is NULL.

1965           GrB\_DOMAIN\_MISMATCH The domains of the `v` vector or `values` array are incompatible with  
1966                   one another.

## 1967 Description

1968 This method will extract all the tuples from the GraphBLAS vector `v`. The values associated  
1969 with those tuples are placed in the `values` array and the indices are placed in the `indices` array.  
1970 Both `indices` and `values` must be pre-allocated by the user to have enough space to hold at least  
1971 GrB\_Vector\_nvals(`v`) elements before calling this function.

1972 Upon return of this function, `n` will be set to the number of values (and indices) copied. Also, the  
1973 entries of `indices` are unique, but not necessarily sorted. Each tuple  $(i, v_i)$  in `v` is unzipped and  
1974 copied into a distinct  $k$ th location in output vectors:

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

1975 where  $0 \leq k < \text{GrB\_Vector\_nvals}(v)$ . No gaps in output vectors are allowed; that is, if `indices`[ $k$ ]  
1976 and `values`[ $k$ ] exist upon return, so does `indices`[ $j$ ] and `values`[ $j$ ] for all  $j$  such that  $0 \leq j < k$ .

1977 Note that if the value in `n` on input is less than the number of values contained in the vector `v`,  
1978 then a GrB\_INSUFFICIENT\_SPACE error is returned because it is undefined which subset of values  
1979 would be extracted otherwise.

1980 In both GrB\_BLOCKING mode GrB\_NONBLOCKING mode if the method exits with return value  
1981 GrB\_SUCCESS, the new contents of the arrays `indices` and `values` are as defined above.

## 1982 4.2.5 Matrix methods

### 1983 4.2.5.1 Matrix\_new: Construct new matrix

1984 Creates a new matrix with specified domain and dimensions.

## 1985 C Syntax

```
1986         GrB_Info GrB_Matrix_new(GrB_Matrix *A,  
1987                                 GrB_Type      d,  
1988                                 GrB_Index     nrows,  
1989                                 GrB_Index     ncols);
```

## 1990 Parameters

1991 **A** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1992 matrix.

1993 **d** (IN) The type corresponding to the domain of the matrix being created. Can be  
1994 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
1995 GraphBLAS type.

1996 **nrows** (IN) The number of rows of the matrix being created.

1997 **ncols** (IN) The number of columns of the matrix being created.

## 1998 Return Values

1999 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2000 blocking mode, this indicates that the API checks for the input ar-  
2001 guments passed successfully. Either way, output matrix **A** is ready  
2002 to be used in the next method of the sequence.

2003 **GrB\_PANIC** Unknown internal error.

2004 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2005 GraphBLAS objects (input or output) is in an invalid state caused  
2006 by a previous execution error. Call **GrB\_error()** to access any error  
2007 messages generated by the implementation.

2008 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2009 **GrB\_UNINITIALIZED\_OBJECT** The **GrB\_Type** object has not been initialized by a call to **GrB\_Type\_new**  
2010 (needed for user-defined types).

2011 **GrB\_NULL\_POINTER** The **A** pointer is **NULL**.

2012 **GrB\_INVALID\_VALUE** **nrows** or **ncols** is zero or outside the range of the type **GrB\_Index**.

## 2013 Description

2014 Creates a new matrix **A** of domain **D**(**d**), size **nrows**  $\times$  **ncols**, and empty **L**(**A**). The method returns  
2015 a handle to the new matrix in **A**.

2016 It is not an error to call this method more than once on the same variable; however, the handle to  
2017 the previously created object will be overwritten.

#### 2018 4.2.5.2 Matrix\_dup: Construct a copy of a GraphBLAS matrix

2019 Creates a new matrix with the same domain, dimensions, and contents as another matrix.

#### 2020 C Syntax

```
2021         GrB_Info GrB_Matrix_dup(GrB_Matrix      *C,  
2022                                const GrB_Matrix A);
```

#### 2023 Parameters

2024 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2025 matrix.

2026 A (IN) The GraphBLAS matrix to be duplicated.

#### 2027 Return Values

2028 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2029 blocking mode, this indicates that the API checks for the input  
2030 arguments passed successfully. Either way, output matrix C is ready  
2031 to be used in the next method of the sequence.

2032 GrB\_PANIC Unknown internal error.

2033 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2034 GraphBLAS objects (input or output) is in an invalid state caused  
2035 by a previous execution error. Call GrB\_error() to access any error  
2036 messages generated by the implementation.

2037 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2038 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2039 any matrix constructor.

2040 GrB\_NULL\_POINTER The C pointer is NULL.

#### 2041 Description

2042 Creates a new matrix C of domain D(A), size nrows(A) × ncols(A), and contents L(A). It returns  
2043 a handle to it in C.



2044 It is not an error to call this method more than once on the same variable; however, the handle to  
2045 the previously created object will be overwritten.

#### 2046 4.2.5.3 Matrix\_diag: Construct a diagonal GraphBLAS matrix

2047 Creates a new matrix with the same domain and contents as a GrB\_Vector, and square dimensions  
2048 appropriate for placing the contents of the vector along the specified diagonal of the matrix.

#### 2049 C Syntax

```
2050         GrB_Info GrB_Matrix_diag(GrB_Matrix      *C,  
2051                                 const GrB_Vector  v,  
2052                                 int64_t           k);
```

#### 2053 Parameters

2054 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2055 matrix. The matrix is square with each dimension equal to  $\text{size}(\mathbf{v}) + |k|$ .

2056 v (IN) The GraphBLAS vector whose contents will be copied to the diagonal of the  
2057 matrix.

2058 k (IN) The diagonal to which the vector is assigned.  $k = 0$  represents the main  
2059 diagonal,  $k > 0$  is above the main diagonal, and  $k < 0$  is below.

#### 2060 Return Values

2061 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2062 blocking mode, this indicates that the API checks for the input  
2063 arguments passed successfully. Either way, output matrix C is ready  
2064 to be used in the next method of the sequence.

2065 GrB\_PANIC Unknown internal error.

2066 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2067 GraphBLAS objects (input or output) is in an invalid state caused  
2068 by a previous execution error. Call GrB\_error() to access any error  
2069 messages generated by the implementation.

2070 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

2071 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, v, has not been initialized by a call to  
2072 Vector\_new or Vector\_dup.

2073 GrB\_NULL\_POINTER The C pointer is NULL.

## 2074 Description

2075 Creates a new matrix **C** of domain **D(v)**, size  $(\mathbf{size}(\mathbf{v}) + |k|) \times (\mathbf{size}(\mathbf{v}) + |k|)$ , and contents

$$\begin{aligned} 2076 \quad \mathbf{L}(\mathbf{C}) &= \{(i, i + k, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k \geq 0 \text{ or} \\ 2077 \quad \mathbf{L}(\mathbf{C}) &= \{(i - k, i, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k < 0. \end{aligned}$$

2078 It returns a handle to it in **C**. It is not an error to call this method more than once on the same  
2079 variable; however, the handle to the previously created object will be overwritten.

## 2080 4.2.5.4 Matrix\_resize: Resize a matrix

2081 Changes the dimensions of an existing matrix.

## 2082 C Syntax

```
2083      GrB_Info GrB_Matrix_resize(GrB_Matrix C,  
2084                               GrB_Index  nrows,  
2085                               GrB_Index  ncols);
```

## 2086 Parameters

2087 **C** (INOUT) An existing Matrix object that is being resized.

2088 **nrows** (IN) The new number of rows of the matrix. It can be smaller or larger than the  
2089 current number of rows.

2090 **ncols** (IN) The new number of columns of the matrix. It can be smaller or larger than  
2091 the current number of columns.

## 2092 Return Values

2093 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2094 blocking mode, this indicates that the API checks for the input  
2095 arguments passed successfully. Either way, output matrix **C** is ready  
2096 to be used in the next method of the sequence.

2097 **GrB\_PANIC** Unknown internal error.

2098 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2099 GraphBLAS objects (input or output) is in an invalid state caused  
2100 by a previous execution error. Call **GrB\_error()** to access any error  
2101 messages generated by the implementation.

2102 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2103           GrB\_NULL\_POINTER The C pointer is NULL.

2104           GrB\_INVALID\_VALUE nrows or ncols is zero or outside the range of the type GrB\_Index.

## 2105   **Description**

2106   Changes the number of rows and columns of C to nrows and ncols, respectively. The domain  $\mathbf{D}(\mathbf{C})$   
2107   of matrix C remains the same. The contents  $\mathbf{L}(\mathbf{C})$  are modified as described below.

2108   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  
2109   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  
2110   greater than or equal to ncols are dropped.  
2111

### 2112   **4.2.5.5   Matrix\_clear: Clear a matrix**

2113   Removes all elements (tuples) from a matrix.

## 2114   **C Syntax**

2115           GrB\_Info GrB\_Matrix\_clear(GrB\_Matrix A);

## 2116   **Parameters**

2117           A (IN) An existing GraphBLAS matrix to clear.

## 2118   **Return Values**

2119           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2120                       blocking mode, this indicates that the API checks for the input ar-  
2121                       guments passed successfully. Either way, output matrix A is ready  
2122                       to be used in the next method of the sequence.

2123           GrB\_PANIC Unknown internal error.

2124           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2125                       GraphBLAS objects (input or output) is in an invalid state caused  
2126                       by a previous execution error. Call GrB\_error() to access any error  
2127                       messages generated by the implementation.

2128           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2129           GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2130                       any matrix constructor.

## 2131 Description

2132 Removes all elements (tuples) from an existing matrix. After the call to `GrB_Matrix_clear(A)`,  
2133  $\mathbf{L}(\mathbf{A}) = \emptyset$ . The dimensions of the matrix do not change.

## 2134 4.2.5.6 Matrix\_nrows: Number of rows in a matrix

2135 Retrieve the number of rows in a matrix.

## 2136 C Syntax

```
2137         GrB_Info GrB_Matrix_nrows(GrB_Index      *nrows,  
2138                                   const GrB_Matrix A);
```

## 2139 Parameters

2140 nrows (OUT) On successful return, contains the number of rows in the matrix.

2141 A (IN) An existing GraphBLAS matrix being queried.

## 2142 Return Values

2143 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
2144 cessfully and the value of `nrows` has been set.

2145 GrB\_PANIC Unknown internal error.

2146 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2147 GraphBLAS objects (input or output) is in an invalid state caused  
2148 by a previous execution error. Call `GrB_error()` to access any error  
2149 messages generated by the implementation.

2150 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to  
2151 any matrix constructor.

2152 GrB\_NULL\_POINTER `nrows` pointer is NULL.

## 2153 Description

2154 Return `nrows(A)` in `nrows` (the number of rows).

## 2155 4.2.5.7 Matrix\_ncols: Number of columns in a matrix

2156 Retrieve the number of columns in a matrix.

## 2157 C Syntax

```
2158         GrB_Info GrB_Matrix_ncols(GrB_Index      *ncols,  
2159                                   const GrB_Matrix A);
```

## 2160 Parameters

2161 ncols (OUT) On successful return, contains the number of columns in the matrix.

2162 A (IN) An existing GraphBLAS matrix being queried.

## 2163 Return Values

2164 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
2165 cessfully and the value of ncols has been set.

2166 GrB\_PANIC Unknown internal error.

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

2171 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2172 any matrix constructor.

2173 GrB\_NULL\_POINTER ncols pointer is NULL.

## 2174 Description

2175 Return **ncols(A)** in ncols (the number of columns).

## 2176 4.2.5.8 Matrix\_nvals: Number of stored elements in a matrix

2177 Retrieve the number of stored elements (tuples) in a matrix.

## 2178 C Syntax

```
2179         GrB_Info GrB_Matrix_nvals(GrB_Index      *nvals,  
2180                                   const GrB_Matrix A);
```

2181 **Parameters**

2182            **nvals** (OUT) On successful return, contains the number of stored elements (tuples) in  
2183            the matrix.

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

2185 **Return Values**

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

2188            **GrB\_PANIC** Unknown internal error.

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

2193            **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2194            **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to  
2195            any matrix constructor.

2196            **GrB\_NULL\_POINTER** The **nvals** pointer is **NULL**.

2197 **Description**

2198            Return **nvals(A)** in **nvals**. This is the number of tuples stored in matrix **A**, which is the size of  
2199            **L(A)** (see Section 3.5.3).

2200 **4.2.5.9 Matrix\_build: Store elements from tuples into a matrix**

2201 **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);
```

2202 **Parameters**

2203            **C** (INOUT) An existing Matrix object to store the result.

2204 **row\_indices** (IN) Pointer to an array of row indices.

2205 **col\_indices** (IN) Pointer to an array of column indices.

2206 **values** (IN) Pointer to an array of scalars of a type that is compatible with the domain of  
2207 matrix, **C**.

2208 **n** (IN) The number of entries contained in each array (the same for **row\_indices**,  
2209 **col\_indices**, and **values**).

2210 **dup** (IN) An associative and commutative binary operator to apply when duplicate  
2211 values for the same location are present in the input arrays. All three domains of  
2212 **dup** must be the same; hence  $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$ . If **dup** is **GrB\_NULL**,  
2213 then duplicate locations will result in an error.

## 2214 Return Values

2215 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2216 blocking mode, this indicates that the API checks for the input  
2217 arguments passed successfully. Either way, output matrix **C** is  
2218 ready to be used in the next method of the sequence.

2219 **GrB\_PANIC** Unknown internal error.

2220 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
2221 opaque GraphBLAS objects (input or output) is in an invalid  
2222 state caused by a previous execution error. Call **GrB\_error()** to  
2223 access any error messages generated by the implementation.

2224 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2225 **GrB\_UNINITIALIZED\_OBJECT** Either **C** has not been initialized by a call to any matrix construc-  
2226 tor, or **dup** has not been initialized by a call to **GrB\_BinaryOp\_new**.

2227 **GrB\_NULL\_POINTER** **row\_indices**, **col\_indices** or **values** pointer is **NULL**.

2228 **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **row\_indices** or **col\_indices** is outside the allowed range  
2229 for **C**.

2230 **GrB\_DOMAIN\_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are  
2231 not all the same, or the domains of **values** and **C** are incompatible  
2232 with each other or  $D_{dup}$ .

2233 **GrB\_OUTPUT\_NOT\_EMPTY** Output matrix **C** already contains valid tuples (elements). In  
2234 other words, **GrB\_Matrix\_nvals(C)** returns a positive value.

2235 **GrB\_INVALID\_VALUE** **indices** contains a duplicate location and **dup** is **GrB\_NULL**.

## 2236 Description

2237 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  
 2238 only differs from  $\mathbf{C}$  in its domain; otherwise,  $\tilde{\mathbf{C}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \emptyset \rangle$ .

2239 Each tuple  $\{\text{row\_indices}[k], \text{col\_indices}[k], \text{values}[k]\}$ , where  $0 \leq k < n$ , is a contribution to the  
 2240 output in the form of

$$2241 \quad \tilde{\mathbf{C}}(\text{row\_indices}[k], \text{col\_indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \text{dup} \neq \text{GrB\_NULL} \\ (\mathbf{D}(\mathbf{C})) \text{values}[k] & \text{otherwise.} \end{cases}$$

2242 If multiple values for the same location are present in the input arrays and `dup` is not `GrB_NULL`,  
 2243 `dup` is used to reduce the values before assignment into  $\tilde{\mathbf{C}}$  as follows:

$$2244 \quad \tilde{\mathbf{C}}_{ij} = \bigoplus_{k: \text{row\_indices}[k]=i \wedge \text{col\_indices}[k]=j} (D_{dup}) \text{values}[k],$$

2245 where  $\oplus$  is the `dup` binary operator. Finally, the resulting  $\tilde{\mathbf{C}}$  is copied into  $\mathbf{C}$  via typecasting its  
 2246 values to  $\mathbf{D}(\mathbf{C})$  if necessary. If  $\oplus$  is not associative or not commutative, the result is undefined.

2247 The nonopaque input arrays `row_indices`, `col_indices`, and `values` must be at least as large as `n`.

2248 It is an error to call this function on an output object with existing elements. In other words,  
 2249 `GrB_Matrix_nvals(C)` should evaluate to zero prior to calling this function.

2250 After `GrB_Matrix_build` returns, it is safe for a programmer to modify or delete the arrays `row_indices`,  
 2251 `col_indices`, or `values`.

### 2252 4.2.5.10 Matrix\_setElement: Set a single element in matrix

2253 Set one element of a matrix to a given value.

## 2254 C Syntax

```
2255 // scalar value
2256 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2257                                <type>         val,
2258                                GrB_Index        row_index,
2259                                GrB_Index        col_index);
2260
2261 // GraphBLAS scalar
2262 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2263                                const GrB_Scalar s,
2264                                GrB_Index        row_index,
2265                                GrB_Index        col_index);
```



## 2266 Parameters

2267           **C** (INOUT) An existing GraphBLAS matrix for which an element is to be assigned.

2268           **val** or **s** (IN) Scalar to assign. Its domain (type) must be compatible with the domain of

2269           **C**.

2270           **row\_index** (IN) Row index of element to be assigned

2271           **col\_index** (IN) Column index of element to be assigned

## 2272 Return Values

2273           **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-

2274           blocking mode, this indicates that the compatibility tests on in-

2275           dex/dimensions and domains for the input arguments passed suc-

2276           cessfully. Either way, the output matrix **C** is ready to be used in

2277           the next method of the sequence.

2278           **GrB\_PANIC** Unknown internal error.

2279           **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque

2280           GraphBLAS objects (input or output) is in an invalid state caused

2281           by a previous execution error. Call **GrB\_error()** to access any error

2282           messages generated by the implementation.

2283           **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2284           **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, or GraphBLAS scalar, **s**, has not been

2285           initialized by a call to a respective constructor.

2286           **GrB\_INVALID\_INDEX** **row\_index** or **col\_index** is outside the allowable range (i.e., not less

2287           than **nrows(C)** or **ncols(C)**, respectively).

2288           **GrB\_DOMAIN\_MISMATCH** The domains of the matrix and the scalar are incompatible.

## 2289 Description

2290 First, the scalar and output matrix are tested for domain compatibility as follows: **D(val)** or

2291 **D(s)** must be compatible with **D(C)**. Two domains are compatible with each other if values from

2292 one domain can be cast to values in the other domain as per the rules of the C language. In

2293 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-

2294 defined type is only compatible with itself. If any compatibility rule above is violated, execution of

2295 **GrB\_Matrix\_setElement** ends and the domain mismatch error listed above is returned.

2296 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2297 \quad & 0 \leq \text{row\_index} < \text{nrows}(\mathbf{C}), \\ & 0 \leq \text{col\_index} < \text{ncols}(\mathbf{C}) \end{aligned}$$

2298 If either of these conditions is violated, execution of `GrB_Matrix_setElement` ends and the invalid  
 2299 index error listed above is returned.

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

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

2300 In the case of a transparent scalar or if  $\mathbf{L}(s)$  is not empty, then a value will be stored at the  
 2301 specified location in  $C$ , overwriting any value that may have been stored there before. In the case  
 2302 of a GraphBLAS scalar and if  $\mathbf{L}(s)$  is empty, then any value stored at the specified location in  $C$   
 2303 will be removed.

2304 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents  
 2305 of  $C$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with  
 2306 return value `GrB_SUCCESS` and the new content of vector  $C$  is as defined above but may not be  
 2307 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 2308 **4.2.5.11 Matrix\_removeElement: Remove an element from a matrix**

2309 Remove (annihilate) one stored element from a matrix.

#### 2310 **C Syntax**

```
2311      GrB_Info GrB_Matrix_removeElement(GrB_Matrix  C,
2312                                         GrB_Index   row_index,
2313                                         GrB_Index   col_index);
```

#### 2314 **Parameters**

2315 `C` (INOUT) An existing GraphBLAS matrix from which an element is to be removed.

2316 `row_index` (IN) Row index of element to be removed

2317 `col_index` (IN) Column index of element to be removed

#### 2318 **Return Values**

2319 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
 2320 blocking mode, this indicates that the compatibility tests on in-  
 2321 dex/dimensions and domains for the input arguments passed suc-  
 2322 cessfully. Either way, the output matrix  $C$  is ready to be used in  
 2323 the next method of the sequence.

2324 `GrB_PANIC` Unknown internal error.

2325       GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 2326       GraphBLAS objects (input or output) is in an invalid state caused  
 2327       by a previous execution error. Call GrB\_error() to access any error  
 2328       messages generated by the implementation.

2329       GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2330 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, C, has not been initialized by a call to  
 2331       any matrix constructor.

2332       GrB\_INVALID\_INDEX row\_index or col\_index is outside the allowable range (i.e., not less  
 2333       than nrows(C) or ncols(C), respectively).

## 2334 Description

2335 First, both index parameters are checked for valid values where following conditions must hold:

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

2337 If either of these conditions is violated, execution of GrB\_Matrix\_removeElement ends and the  
 2338 invalid index error listed above is returned.

2339 We are now ready to carry out the removal of a value that may be stored at the location specified by  
 2340 (row\_index, col\_index). If a value does not exist at the specified location in C, no error is reported  
 2341 and the operation has no effect on the state of C. In either case, the following will be true on return  
 2342 from this method: (row\_index, col\_index)  $\notin$  ind(C)

2343 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new contents  
 2344 of C is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method exits with  
 2345 return value GrB\_SUCCESS and the new content of vector C is as defined above but may not be  
 2346 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 2347 4.2.5.12 Matrix\_extractElement: Extract a single element from a matrix

2348 Extract one element of a matrix into a scalar.

## 2349 C Syntax

```

2350     // scalar value
2351     GrB_Info GrB_Matrix_extractElement(<type>          *val,
2352                                     const GrB_Matrix  A,
2353                                     GrB_Index          row_index,
2354                                     GrB_Index          col_index);
2355
2356     // GraphBLAS scalar
  
```

```

2357         GrB_Info GrB_Matrix_extractElement(GrB_Scalar      s,
2358                                           const GrB_Matrix A,
2359                                           GrB_Index      row_index,
2360                                           GrB_Index      col_index);
2361

```

## 2362 Parameters

2363     **val or s (INOUT)** An existing scalar whose domain is compatible with the domain of matrix  
2364     A. On successful return, this scalar holds the result of the extract. Any previous  
2365     value stored in **val** or **s** is overwritten.

2366     **A (IN)** The GraphBLAS matrix from which an element is extracted.

2367     **row\_index (IN)** The row index of location in **A** to extract.

2368     **col\_index (IN)** The column index of location in **A** to extract.

## 2369 Return Values

2370     **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2371     cessfully. This indicates that the compatibility tests on dimensions  
2372     and domains for the input arguments passed successfully, and the  
2373     output scalar, **val** or **s**, has been computed and is ready to be used  
2374     in the next method of the sequence.

2375     **GrB\_NO\_VALUE** When using the transparent scalar, **val**, this is returned when there  
2376     is no stored value at specified location.

2377     **GrB\_PANIC** Unknown internal error.

2378     **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2379     GraphBLAS objects (input or output) is in an invalid state caused  
2380     by a previous execution error. Call **GrB\_error()** to access any error  
2381     messages generated by the implementation.

2382     **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2383     **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, or scalar, **s**, has not been initialized by  
2384     a call to a corresponding constructor.

2385     **GrB\_NULL\_POINTER** **val** pointer is **NULL**.

2386     **GrB\_INVALID\_INDEX** **row\_index** or **col\_index** is outside the allowable range (i.e. less than  
2387     zero or greater than or equal to **nrows(A)** or **ncols(A)**, respec-  
2388     tively).

2389     **GrB\_DOMAIN\_MISMATCH** The domains of the matrix and scalar are incompatible.

## 2390 Description

2391 First, the scalar and input matrix are tested for domain compatibility as follows:  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(\mathbf{s})$   
 2392 must be compatible with  $\mathbf{D}(\mathbf{A})$ . Two domains are compatible with each other if values from  
 2393 one domain can be cast to values in the other domain as per the rules of the C language. In  
 2394 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
 2395 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
 2396 `GrB_Matrix_extractElement` ends and the domain mismatch error listed above is returned.

2397 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2398 \quad & 0 \leq \text{row\_index} < \mathbf{nrows}(\mathbf{A}), \\ & 0 \leq \text{col\_index} < \mathbf{ncols}(\mathbf{A}) \end{aligned}$$

2399 If either condition is violated, execution of `GrB_Matrix_extractElement` ends and the invalid index  
 2400 error listed above is returned.

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

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

2401 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  $\text{val}$   
 2402 with casting as necessary. If  $(\text{row\_index}, \text{col\_index}) \notin \mathbf{ind}(\mathbf{A})$ , then one of the follow occurs  
 2403 depending on output scalar type:

- 2404 • The GraphBLAS scalar,  $\mathbf{s}$ , is cleared and `GrB_SUCCESS` is returned.
- 2405 • The non-opaque scalar,  $\text{val}$ , is unchanged, and `GrB_NO_VALUE` is returned.

2406 When using the non-opaque scalar variant ( $\text{val}$ ) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`  
 2407 mode, the new contents of  $\text{val}$  are as defined above if the method exits with return value `GrB_SUCCESS`  
 2408 or `GrB_NO_VALUE`.

2409 When using the GraphBLAS scalar variant ( $\mathbf{s}$ ) with a `GrB_SUCCESS` return value, the method  
 2410 exits and the new contents of  $\mathbf{s}$  is as defined above and fully computed in `GrB_BLOCKING` mode.  
 2411 In `GrB_NONBLOCKING` mode, the new contents of  $\mathbf{s}$  is as defined above but may not be fully  
 2412 computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 2413 4.2.5.13 Matrix\_extractTuples: Extract tuples from a matrix

2414 Extract the contents of a GraphBLAS matrix into non-opaque data structures.

## 2415 C Syntax

```
2416      GrB_Info GrB_Matrix_extractTuples(GrB_Index      *row_indices,
2417                                       GrB_Index      *col_indices,
```

```

2418                                     <type>          *values,
2419                                     GrB_Index         *n,
2420                                     const GrB_Matrix   A);

```

## 2421 Parameters

2422     **row\_indices** (OUT) Pointer to an array of row indices that is large enough to hold all of the  
2423     row indices.

2424     **col\_indices** (OUT) Pointer to an array of column indices that is large enough to hold all of the  
2425     column indices.

2426     **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of  
2427     the stored values whose type is compatible with  $\mathbf{D}(\mathbf{A})$ .

2428     **n** (INOUT) Pointer to a value indicating (in input) the number of elements the **values**,  
2429     **row\_indices**, and **col\_indices** arrays can hold. Upon return, it will contain the  
2430     number of values written to the arrays.

2431     **A** (IN) An existing GraphBLAS matrix.

## 2432 Return Values

2433     **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2434     cessfully. This indicates that the compatibility tests on the input  
2435     argument passed successfully, and the output arrays, **indices** and  
2436     **values**, have been computed.

2437     **GrB\_PANIC** Unknown internal error.

2438     **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2439     GraphBLAS objects (input or output) is in an invalid state caused  
2440     by a previous execution error. Call **GrB\_error()** to access any error  
2441     messages generated by the implementation.

2442     **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2443     **GrB\_INSUFFICIENT\_SPACE** Not enough space in **row\_indices**, **col\_indices**, and **values** (as indi-  
2444     cated by the **n** parameter) to hold all of the tuples that will be  
2445     extracted.

2446     **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to  
2447     any matrix constructor.

2448     **GrB\_NULL\_POINTER** **row\_indices**, **col\_indices**, **values** or **n** pointer is NULL.

2449     **GrB\_DOMAIN\_MISMATCH** The domains of the **A** matrix and **values** array are incompatible  
2450     with one another.

## 2451 Description

2452 This method will extract all the tuples from the GraphBLAS matrix **A**. The values associated with  
2453 those tuples are placed in the **values** array, the column indices are placed in the **col\_indices** array,  
2454 and the row indices are placed in the **row\_indices** array. These output arrays are pre-allocated by  
2455 the user before calling this function such that each output array has enough space to hold at least  
2456 **GrB\_Matrix\_nvals(A)** elements.

2457 Upon return of this function, a pair of  $\{\text{row\_indices}[k], \text{col\_indices}[k]\}$  are unique for every valid  
2458  $k$ , but they are not required to be sorted in any particular order. Each tuple  $(i, j, A_{ij})$  in **A** is  
2459 unzipped and copied into a distinct  $k$ th location in output vectors:

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

2460 where  $0 \leq k < \text{GrB\_Matrix\_nvals}(v)$ . No gaps in output vectors are allowed; that is, if **row\_indices**[ $k$ ],  
2461 **col\_indices**[ $k$ ] and **values**[ $k$ ] exist upon return, so does **row\_indices**[ $j$ ], **col\_indices**[ $j$ ] and **values**[ $j$ ] for  
2462 all  $j$  such that  $0 \leq j < k$ .

2463 Note that if the value in **n** on input is less than the number of values contained in the matrix **A**,  
2464 then a **GrB\_INSUFFICIENT\_SPACE** error is returned since it is undefined which subset of values  
2465 would be extracted.

2466 In both **GrB\_BLOCKING** mode **GrB\_NONBLOCKING** mode if the method exits with return value  
2467 **GrB\_SUCCESS**, the new contents of the arrays **row\_indices**, **col\_indices** and **values** are as defined  
2468 above.

2469 **4.2.5.14 Matrix\_exportHint: Provide a hint as to which storage format might be most**  
2470 **efficient for exporting a matrix**

## 2471 C Syntax

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

## 2472 Parameters

2473 **hint** (OUT) Pointer to a value of type **GrB\_Format**.

2474 **A** (IN) A GraphBLAS matrix object.

## 2475 Return Values

2476 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2477 cessfully and the value of **hint** has been set.

2478 **GrB\_PANIC** Unknown internal error.

2479           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
 2480                                   opaque GraphBLAS objects (input or output) is in an invalid  
 2481                                   state caused by a previous execution error. Call GrB\_error() to  
 2482                                   access any error messages generated by the implementation.

2483           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2484           GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
 2485                                   any matrix constructor.

2486           GrB\_NULL\_POINTER hint is NULL.

2487           GrB\_NO\_VALUE If the implementation does not have a preferred format, it may  
 2488                                   return the value GrB\_NO\_VALUE.

## 2489 Description

2490 Given a GraphBLAS matrix A, provide a hint as to which format might be most efficient for  
 2491 exporting the matrix A. GraphBLAS implementations might return the current storage format of  
 2492 the matrix, or the format to which it could most efficiently be exported. However, implementations  
 2493 are free to return any value for format defined in Section 3.5.3.1. Note that an implementation is  
 2494 free to refuse to provide a format hint, returning GrB\_NO\_VALUE.

### 2495 4.2.5.15 Matrix\_exportSize: Return the array sizes necessary to export a GraphBLAS 2496 matrix object

## 2497 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);

```

## 2498 Parameters

2499           n\_indptr (OUT) Pointer to a value of type GrB\_Index.

2500           n\_indices (OUT) Pointer to a value of type GrB\_Index.

2501           n\_values (OUT) Pointer to a value of type GrB\_Index.

2502           format (IN) a value indicating the format in which the matrix will be exported, as defined  
 2503                                   in Section 3.5.3.1.

2504           A (IN) A GraphBLAS matrix object.



## 2505 Return Values

2506                   GrB\_SUCCESS In blocking mode or non-blocking mode, the operation com-  
2507                                   pleted successfully. This indicates that the API checks for the  
2508                                   input arguments passed successfully, and the number of elements  
2509                                   necessary for the export buffers have been written to `n_indptr`,  
2510                                   `n_indices`, and `n_values`, respectively.

2511                   GrB\_PANIC Unknown internal error.

2512                   GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
2513                                   opaque GraphBLAS objects (input or output) is in an invalid  
2514                                   state caused by a previous execution error. Call `GrB_error()` to  
2515                                   access any error messages generated by the implementation.

2516                   GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2517                   GrB\_UNINITIALIZED\_OBJECT The GraphBLAS Matrix, `A`, has not been initialized by a call to  
2518                                   any matrix constructor.

2519                   GrB\_NULL\_POINTER `n_indptr`, `n_indices`, or `n_values` is NULL.

## 2520 Description

2521 Given a matrix `A`, returns the required capacities of arrays `values`, `indptr`, and `indices` necessary to  
2522 export the matrix in the format specified by `format`. The output values `n_values`, `n_indptr`, and  
2523 `indices` will contain the corresponding sizes of the arrays (in number of elements) that must be  
2524 allocated to hold the exported matrix. The argument `format` can be chosen arbitrarily by the user  
2525 as one of the values defined in Section 3.5.3.1.

### 2526 4.2.5.16 Matrix\_export: Export a GraphBLAS matrix to a pre-defined format

#### 2527 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);
```

## 2528 Parameters

- 2529        **indptr** (INOUT) Pointer to an array that will hold row or column offsets, or row in-  
2530        dices, depending on the value of **format**. It must be large enough to hold at  
2531        least **n\_indptr** elements of type **GrB\_Index**, where **n\_indices** was returned from  
2532        **GrB\_Matrix\_exportSize()** method.
- 2533        **indices** (INOUT) Pointer to an array that will hold row or column indices of the elements  
2534        in **values**, depending on the value of **format**. It must be large enough to hold at  
2535        least **n\_indices** elements of type **GrB\_Index**, where **n\_indices** was returned from  
2536        **GrB\_Matrix\_exportSize()** method.
- 2537        **values** (INOUT) Pointer to an array that will hold stored values. The type of ele-  
2538        ment must match the type of the values stored in **A**. It must be large enough  
2539        to hold at least **n\_values** elements of that type, where **n\_values** was returned from  
2540        **GrB\_Matrix\_exportSize**.
- 2541        **n\_indptr** (INOUT) Pointer to a value indicating (on input) the number of elements the **indptr**  
2542        array can hold. Upon return, it will contain the number of elements written to the  
2543        array.
- 2544        **n\_indices** (INOUT) Pointer to a value indicating (on input) the number of elements the **indices**  
2545        array can hold. Upon return, it will contain the number of elements written to the  
2546        array.
- 2547        **n\_values** (INOUT) Pointer to a value indicating (on input) the number of elements the **values**  
2548        array can hold. Upon return, it will contain the number of elements written to the  
2549        array.
- 2550        **format** (IN) a value indicating the format in which the matrix will be exported, as defined  
2551        in Section 3.5.3.1.
- 2552        **A** (IN) A GraphBLAS matrix object.

## 2553 Return Values

- 2554        **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2555        cessfully. This indicates that the compatibility tests on the input  
2556        argument passed successfully, and the output arrays, **indptr**, **in-**  
2557        **dices** and **values**, have been computed.
- 2558        **GrB\_PANIC** Unknown internal error.
- 2559        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
2560        opaque GraphBLAS objects (input or output) is in an invalid  
2561        state caused by a previous execution error. Call **GrB\_error()** to  
2562        access any error messages generated by the implementation.
- 2563        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.



2588        **nrows** (IN) Integer value holding the number of rows in the matrix.

2589        **ncols** (IN) Integer value holding the number of columns in the matrix.

2590        **indptr** (IN) Pointer to an array of row or column offsets, or row indices, depending on the  
2591        value of **format**.

2592        **indices** (IN) Pointer to an array row or column indices of the elements in **values**, depending  
2593        on the value of **format**.

2594        **values** (IN) Pointer to an array of values. Type must match the type of **d**.

2595        **n\_indptr** (IN) Integer value holding the number of elements in the array pointed to by **indptr**.

2596        **n\_indices** (IN) Integer value holding the number of elements in the array pointed to by **indices**.

2597        **n\_values** (IN) Integer value holding the number of elements in the array pointed to by **values**.

2598        **format** (IN) a value indicating the format of the matrix being imported, as defined in  
2599        Section 3.5.3.1.

## 2600 **Return Values**

2601        **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2602        blocking mode, this indicates that the API checks for the input  
2603        arguments passed successfully and the input arrays have been  
2604        consumed. Either way, output matrix **A** is ready to be used in  
2605        the next method of the sequence.

2606        **GrB\_PANIC** Unknown internal error.

2607        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2608        **GrB\_UNINITIALIZED\_OBJECT** The **GrB\_Type** object has not been initialized by a call to **GrB\_Type\_new**  
2609        (needed for user-defined types).

2610        **GrB\_NULL\_POINTER** **A**, **indptr**, **indices** or **values** pointer is **NULL**.

2611        **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **indptr** or **indices** is outside the allowed range for indices  
2612        in **A** and or the size of **values**, **n\_values**, depending on the value  
2613        of **format**.

2614        **GrB\_INVALID\_VALUE** **nrows** or **ncols** is zero or outside the range of the type **GrB\_Index**.

2615        **GrB\_DOMAIN\_MISMATCH** The domain given in parameter **d** does not match the element  
2616        type of **values**.

## 2617 Description

2618 Creates a new matrix **A** of domain **D**(d) and dimension **nrows**  $\times$  **ncols**. The new GraphBLAS  
2619 matrix will be filled with the contents of the matrix pointed to by **indptr**, and **indices**, and **values**.  
2620 The method returns a handle to the new matrix in **A**. The structure of the data being imported is  
2621 defined by **format**, which must be equal to one of the values defined in Section 3.5.3.1. Details of  
2622 the contents of **indptr**, **indices** and **values** for each supported format is given in Appendix B.

2623 It is not an error to call this method more than once on the same output matrix; however, the  
2624 handle to the previously created object will be overwritten.

## 2625 4.2.5.18 Matrix\_serializeSize: Compute the serialize buffer size

2626 Compute the buffer size (in bytes) necessary to serialize a GrB\_Matrix using GrB\_Matrix\_serialize.

## 2627 C Syntax

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

## 2628 Parameters

2629 size (OUT) Pointer to GrB\_Index value where size in bytes of serialized object will be  
2630 written.

2631 A (IN) A GraphBLAS matrix object.

## 2632 Return Values

2633 GrB\_SUCCESS The operation completed successfully and the value pointed to  
2634 by \*size has been computed and is ready to use.

2635 GrB\_PANIC Unknown internal error.

2636 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2637 GrB\_NULL\_POINTER size is NULL.

## 2638 Description

2639 Returns the size in bytes of the data buffer necessary to serialize the GraphBLAS matrix object A.  
2640 Users may then allocate a buffer of size bytes to pass as a parameter to GrB\_Matrix\_serialize.

2641 **4.2.5.19 Matrix\_serialize: Serialize a GraphBLAS matrix.**

2642 Serialize a GraphBLAS Matrix object into an opaque stream of bytes.

2643 **C Syntax**

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

2644 **Parameters**

2645 **serialized\_data** (INOUT) Pointer to the preallocated buffer where the serialized matrix will be  
2646 written.

2647 **serialized\_size** (INOUT) On input, the size in bytes of the buffer pointed to by **serialized\_data**.  
2648 On output, the number of bytes written to **serialized\_data**.

2649 **A** (IN) A GraphBLAS matrix object.

2650 **Return Values**

2651 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2652 cessfully. This indicates that the compatibility tests on the in-  
2653 put argument passed successfully, and the output buffer **serial-  
2654 ized\_data** and **serialized\_size**, have been computed and are ready  
2655 to use.

2656 **GrB\_PANIC** Unknown internal error.

2657 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
2658 opaque GraphBLAS objects (input or output) is in an invalid  
2659 state caused by a previous execution error. Call **GrB\_error()** to  
2660 access any error messages generated by the implementation.

2661 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2662 **GrB\_NULL\_POINTER** **serialized\_data** or **serialize\_size** is NULL.

2663 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to  
2664 any matrix constructor.

2665 **GrB\_INSUFFICIENT\_SPACE** The size of the buffer **serialized\_data** (provided as an input **seri-  
2666 alized\_size**) was not large enough.

## 2667 Description

2668 Serializes a GraphBLAS matrix object to an opaque buffer. To guarantee successful execution,  
2669 the size of the buffer pointed to by `serialized_data`, provided as an input by `serialized_size`, must  
2670 be of at least the number of bytes returned from `GrB_Matrix_serializeSize`. The actual size of the  
2671 serialized matrix written to `serialized_data` is provided upon completion as an output written to  
2672 `serialized_size`.

2673 The contents of the serialized buffer are implementation defined. Thus, a serialized matrix created  
2674 with one library implementation is not necessarily valid for deserialization with another implemen-  
2675 tation.

### 2676 4.2.5.20 Matrix\_deserialize: Deserialize a GraphBLAS matrix.

2677 Construct a new GraphBLAS matrix from a serialized object.

## 2678 C Syntax

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

## 2679 Parameters

2680 A (INOUT) On a successful return, contains a handle to the newly created Graph-  
2681 BLAS matrix.

2682 d (IN) the type of the matrix that was serialized in `serialized_data`.

2683 `serialized_data` (IN) a pointer to a serialized GraphBLAS matrix created with `GrB_Matrix_serialize`.

2684 `serialized_size` (IN) the size of the buffer pointed to by `serialized_data` in bytes.

## 2685 Return Values

2686 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2687 blocking mode, this indicates that the API checks for the input  
2688 arguments passed successfully. Either way, output matrix A is  
2689 ready to be used in the next method of the sequence.

2690 GrB\_PANIC Unknown internal error.

2691 GrB\_INVALID\_OBJECT This is returned if `serialized_data` is invalid or corrupted.

2692 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2693 GrB\_UNINITIALIZED\_OBJECT The GrB\_Type object has not been initialized by a call to GrB\_Type\_new  
2694 (needed for user-defined types).

2695 GrB\_NULL\_POINTER serialized\_data or A is NULL.

2696 GrB\_DOMAIN\_MISMATCH The type given in d does not match the type of the matrix  
2697 serialized in serialized\_data.

## 2698 Description

2699 Creates a new matrix **A** using the serialized matrix object pointed to by `serialized_data`. The object  
2700 pointed to by `serialized_data` must have been created using the method `GrB_Matrix_serialize`. The  
2701 domain of the matrix is given as an input in `d`, which must match the domain of the matrix serialized  
2702 in `serialized_data`. Note that for user-defined types, only the size of the type will be checked.

2703 Since the format of a serialized matrix is implementation-defined, it is not guaranteed that a matrix  
2704 serialized in one library implementation can be deserialized by another.

2705 It is not an error to call this method more than once on the same output matrix; however, the  
2706 handle to the previously created object will be overwritten.

## 2707 4.2.6 Descriptor methods

2708 The methods in this section create and set values in descriptors. A descriptor is an opaque Graph-  
2709 BLAS object the values of which are used to modify the behavior of GraphBLAS operations.

### 2710 4.2.6.1 Descriptor\_new: Create new descriptor

2711 Creates a new (empty or default) descriptor.

## 2712 C Syntax

2713 GrB\_Info GrB\_Descriptor\_new(GrB\_Descriptor \*desc);

## 2714 Parameters

2715 desc (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2716 descriptor.

## 2717 Return Value

2718 GrB\_SUCCESS The method completed successfully.

2719 GrB\_PANIC unknown internal error.



2720        GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

2721        GrB\_NULL\_POINTER desc pointer is NULL.

## 2722    **Description**

2723    Creates a new descriptor object and returns a handle to it in desc. A newly created descriptor can  
2724    be populated by calls to Descriptor\_set.

2725    It is not an error to call this method more than once on the same variable; however, the handle to  
2726    the previously created object will be overwritten.

### 2727    **4.2.6.2    Descriptor\_set: Set content of descriptor**

2728    Sets the content for a field for an existing descriptor.

## 2729    **C Syntax**

```
2730        GrB_Info GrB_Descriptor_set(GrB_Descriptor        desc,  
2731                                    GrB_Desc_Field        field,  
2732                                    GrB_Desc_Value        val);
```

## 2733    **Parameters**

2734        desc (IN) An existing GraphBLAS descriptor to be modified.

2735        field (IN) The field being set.

2736        val (IN) New value for the field being set.

## 2737    **Return Values**

2738        GrB\_SUCCESS operation completed successfully.

2739        GrB\_PANIC unknown internal error.

2740        GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

2741    GrB\_UNINITIALIZED\_OBJECT the desc parameter has not been initialized by a call to new.

2742        GrB\_INVALID\_VALUE invalid value set on the field, or invalid field.

## 2743 Description

2744 For a given descriptor, the `GrB_Descriptor_set` method can be called for each field in the descriptor  
2745 to set the value associated with that field. Valid values for the `field` parameter include the following:

2746 `GrB_OUTP` refers to the output parameter (result) of the operation.

2747 `GrB_MASK` refers to the mask parameter of the operation.

2748 `GrB_INP0` refers to the first input parameters of the operation (matrices and vectors).

2749 `GrB_INP1` refers to the second input parameters of the operation (matrices and vectors).

2750 Valid values for the `val` parameter are:

2751 `GrB_STRUCTURE` Use only the structure of the stored values of the corresponding mask  
2752 (`GrB_MASK`) parameter.

2753 `GrB_COMP` Use the complement of the corresponding mask (`GrB_MASK`) param-  
2754 eter. When combined with `GrB_STRUCTURE`, the complement of the  
2755 structure of the mask is used without evaluating the values stored.

2756 `GrB_TRAN` Use the transpose of the corresponding matrix parameter (valid for input  
2757 matrix parameters only).

2758 `GrB_REPLACE` When assigning the masked values to the output matrix or vector, clear  
2759 the matrix first (or clear the non-masked entries). The default behavior  
2760 is to leave non-masked locations unchanged. Valid for the `GrB_OUTP`  
2761 parameter only.

2762 Descriptor values can only be set, and once set, cannot be cleared. As, in the case of `GrB_MASK`,  
2763 multiple values can be set and all will apply (for example, both `GrB_COMP` and `GrB_STRUCTURE`).  
2764 A value for a given field may be set multiple times but will have no additional effect. Fields that  
2765 have no values set result in their default behavior, as defined in Section 3.7.

## 2766 4.2.7 free: Destroy an object and release its resources

2767 Destroys a previously created GraphBLAS object and releases any resources associated with the  
2768 object.

## 2769 C Syntax

2770 `GrB_Info GrB_free(<GrB_Object> *obj);`

## 2771 Parameters

2772       obj (INOUT) An existing GraphBLAS object to be destroyed. The object must have  
2773       been created by an explicit call to a GraphBLAS constructor. It can be any of the  
2774       opaque GraphBLAS objects such as matrix, vector, descriptor, semiring, monoid,  
2775       binary op, unary op, or type. On successful completion of GrB\_free, obj behaves  
2776       as an uninitialized object.

## 2777 Return Values

2778       GrB\_SUCCESS operation completed successfully

2779       GrB\_PANIC unknown internal error. If this return value is encountered when  
2780       in nonblocking mode, the error responsible for the panic condition  
2781       could be from any method involved in the computation of the input  
2782       object. The GrB\_error() method should be called for additional  
2783       information.

## 2784 Description

2785       GraphBLAS objects consume memory and other resources managed by the GraphBLAS runtime  
2786       system. A call to GrB\_free frees those resources so they are available for use by other GraphBLAS  
2787       objects.

2788       The parameter passed into GrB\_free is a handle referencing a GraphBLAS opaque object of a data  
2789       type from table 2.1. The object must have been created by an explicit call to a GraphBLAS con-  
2790       structor. The behavior of a program that calls GrB\_free on a pre-defined object is implementation  
2791       defined.

2792       After the GrB\_free method returns, the object referenced by the input handle is destroyed and the  
2793       handle has the value GrB\_INVALID\_HANDLE. The handle can be used in subsequent GraphBLAS  
2794       methods but only after the handle has been reinitialized with a call the the appropriate \_new or  
2795       \_dup method.

2796       Note that unlike other GraphBLAS methods, calling GrB\_free with an object with an invalid handle  
2797       is legal. The system may attempt to free resources that might be associated with that object, if  
2798       possible, and return normally.

2799       When using GrB\_free it is possible to create a dangling reference to an object. This would occur  
2800       when a handle is assigned to a second variable of the same opaque type. This creates two handles  
2801       that reference the same object. If GrB\_free is called with one of the variables, the object is destroyed  
2802       and the handle associated with the other variable no longer references a valid object. This is not an  
2803       error condition that the implementation of the GraphBLAS API can be expected to catch, hence  
2804       programmers must take care to prevent this situation from occurring.

2805 **4.2.8 wait: Return once an object is either *complete* or *materialized***

2806 Wait until method calls in a sequence put an object into a state of *completion* or *materialization*.

2807 **C Syntax**

2808 `GrB_Info GrB_wait(GrB_Object obj, GrB_WaitMode mode);`

2809 **Parameters**

2810 `obj` (INOUT) An existing GraphBLAS object. The object must have been created by an  
2811 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS  
2812 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,  
2813 or type. On successful return of `GrB_wait`, the `obj` can be safely read from another  
2814 thread (completion) or all computing to produce `obj` by all GraphBLAS operations  
2815 in its sequence have finished (materialization).

2816 `mode` (IN) Set's the mode for `GrB_wait` for whether it is waiting for `obj` to be in the  
2817 state of *completion* or *materialization*. Acceptable values are `GrB_COMPLETE` or  
2818 `GrB_MATERIALIZE`.

2819 **Return values**

2820 `GrB_SUCCESS` operation completed successfully.

2821 `GrB_INDEX_OUT_OF_BOUNDS` an index out-of-bounds execution error happened during com-  
2822 pletion of pending operations.

2823 `GrB_OUT_OF_MEMORY` and out-of-memory execution error happened during completion  
2824 of pending operations.

2825 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`,  
2826 or other constructor, method.

2827 `GrB_PANIC` unknown internal error.

2828 `GrB_INVALID_VALUE` method called with a `GrB_WaitMode` other than `GrB_COMPLETE`  
2829 `GrB_MATERIALIZE`.

2830 **Description**

2831 On successful return from `GrB_wait()`, the input object, `obj` is in one of two states depending on  
2832 the mode of `GrB_wait`:

- 2833 • *complete*: `obj` can be used in a happens-before relation, so in a properly synchronized program  
2834 it can be safely used as an IN or INOUT parameter in a GraphBLAS method call from another  
2835 thread. This result occurs when the mode parameter is set to `GrB_COMPLETE`.
- 2836 • *materialized*: `obj` is *complete*, but in addition, no further computing will be carried out on  
2837 behalf of `obj` and error information is available. This result occurs when the mode parameter  
2838 is set to `GrB_MATERIALIZE`.

2839 Since in blocking mode OUT or INOUT parameters to any method call are materialized upon return,  
2840 `GrB_wait(obj,mode)` has no effect when called in blocking mode.

2841 In non-blocking mode, the status of any pending method calls, other than those associated with pro-  
2842 ducing the *complete* or *materialized* state of `obj`, are not impacted by the call to `GrB_wait(obj,mode)`.  
2843 Methods in the sequence for `obj`, however, most likely would be impacted by a call to `GrB_wait(obj,mode)`;  
2844 especially in the case of the *materialized* mode for which any computing on behalf of `obj` must be  
2845 finished prior to the return from `GrB_wait(obj,mode)`.

#### 2846 4.2.9 error: Retrieve an error string

2847 Retrieve an error-message about any errors encountered during the processing associated with an  
2848 object.

### 2849 C Syntax

```
2850      GrB_Info GrB_error(const char      **error,
2851                        const GrB_Object  obj);
```

#### 2852 Parameters

2853 `error` (OUT) A pointer to a null-terminated string. The contents of the string are im-  
2854 plementation defined.

2855 `obj` (IN) An existing GraphBLAS object. The object must have been created by an  
2856 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS  
2857 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,  
2858 or type.

#### 2859 Return value

2860 `GrB_SUCCESS` operation completed successfully.

2861 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`,  
2862 or other constructor, method.

2863 `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	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A} \otimes \mathbf{B}$

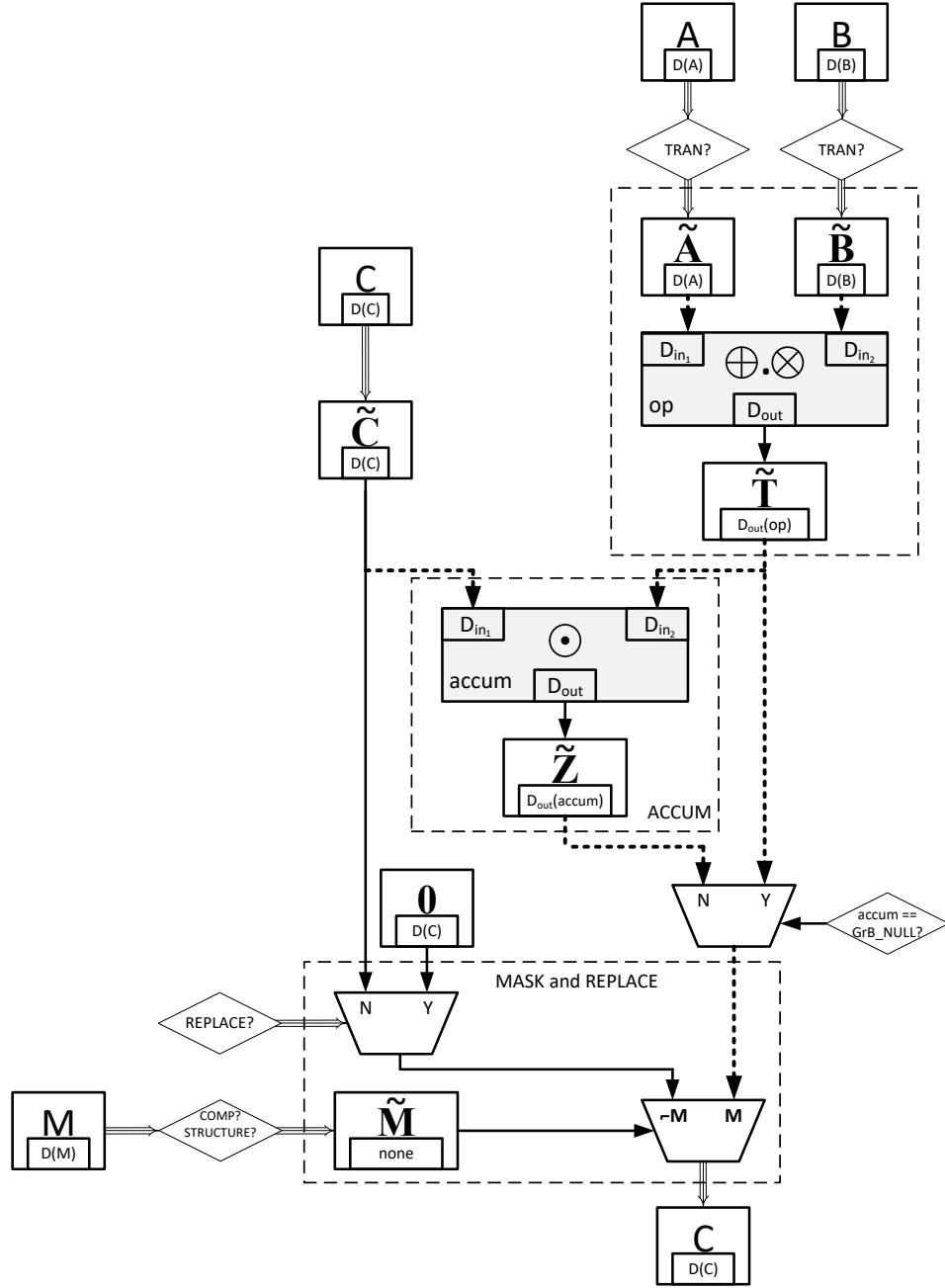


Figure 4.1: Flowchart for the GraphBLAS operations. Although shown specifically for the  $\text{mxm}$  operation, many elements are common to all operations: such as the “ACCUM” and “MASK and REPLACE” blocks. The triple arrows ( $\Rightarrow$ ) denote where “as if copy” takes place (including both collections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

2933 To support the detection of as many cases of uninitialized objects as possible, it is strongly rec-  
 2934 ommended to initialize all GraphBLAS objects to the predefined value `GrB_INVALID_HANDLE` at  
 2935 the point of their declaration, as shown in the following examples:

```
2936      GrB_Type      type = GrB_INVALID_HANDLE;
2937      GrB_Semiring  semiring = GrB_INVALID_HANDLE;
2938      GrB_Matrix    matrix = GrB_INVALID_HANDLE;
```

## 2939 Compliance

2940 We follow a *prescriptive* approach to the definition of the semantics of GraphBLAS operations.  
 2941 That is, for each operation we give a recipe for producing its outcome. Any implementation that  
 2942 produces the same outcome, and follows the GraphBLAS execution model (Section 2.5) and error  
 2943 model (Section 2.6) is a conforming implementation.

### 2944 4.3.1 mxm: Matrix-matrix multiply

2945 Multiplies a matrix with another matrix on a semiring. The result is a matrix.

## 2946 C Syntax

```
2947      GrB_Info GrB_mxm(GrB_Matrix      C,
2948                      const GrB_Matrix Mask,
2949                      const GrB_BinaryOp accum,
2950                      const GrB_Semiring op,
2951                      const GrB_Matrix A,
2952                      const GrB_Matrix B,
2953                      const GrB_Descriptor desc);
```

## 2954 Parameters

2955 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
 2956 that may be accumulated with the result of the matrix product. On output, the  
 2957 matrix holds the results of the operation.

2958 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
 2959 stored into the output matrix C. The mask dimensions must match those of the  
 2960 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
 2961 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types  
 2962 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
 2963 dimensions of C), `GrB_NULL` should be specified.

2964 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
 2965 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
 2966 specified.

2967 **op** (IN) The semiring used in the matrix-matrix multiply.

2968 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
 2969 multiplication.

2970 **B** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
 2971 multiplication.

2972 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
 2973 should be specified. Non-default field/value pairs are listed as follows:  
 2974

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.

## 2976 Return Values

2977 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 2978 blocking mode, this indicates that the compatibility tests on di-  
 2979 mensions and domains for the input arguments passed successfully.  
 2980 Either way, output matrix **C** is ready to be used in the next method  
 2981 of the sequence.

2982 **GrB\_PANIC** Unknown internal error.

2983 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 2984 GraphBLAS objects (input or output) is in an invalid state caused  
 2985 by a previous execution error. Call **GrB\_error()** to access any error  
 2986 messages generated by the implementation.

2987 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

2988 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 2989 a call to **new** (or **Matrix\_dup** for matrix parameters).

2990 **GrB\_DIMENSION\_MISMATCH** Mask and/or matrix dimensions are incompatible.

2991 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
 2992 corresponding domains of the semiring or accumulation operator,  
 2993 or the mask's domain is not compatible with `bool` (in the case where  
 2994 `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 2995 Description

2996 GrB\_mxm computes the matrix product  $C = A \oplus . \otimes B$  or, if an optional binary accumulation operator  
 2997  $(\odot)$  is provided,  $C = C \odot (A \oplus . \otimes B)$  (where matrices  $A$  and  $B$  can be optionally transposed).  
 2998 Logically, this operation occurs in three steps:

2999 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 3000 and dimensions are tested for compatibility.

3001 **Compute** The indicated computations are carried out.

3002 **Output** The result is written into the output matrix, possibly under control of a mask.

3003 Up to four argument matrices are used in the GrB\_mxm operation:

- 3004 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3005 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)
- 3006 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3007 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3008 The argument matrices, the semiring, and the accumulation operator (if provided) are tested for  
 3009 domain compatibility as follows:

- 3010 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 3011 must be from one of the pre-defined types of Table 3.2.
- 3012 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the semiring.
- 3013 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the semiring.
- 3014 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the semiring.
- 3015 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 3016 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
 3017 of the accumulation operator.

3018 Two domains are compatible with each other if values from one domain can be cast to values in  
 3019 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are  
 3020 all compatible with each other. A domain from a user-defined type is only compatible with itself.

3021 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the domain mismatch  
 3022 error listed above is returned.

3023 From the argument matrices, the internal matrices and mask used in the computation are formed  
 3024 ( $\leftarrow$  denotes copy):

- 3025 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 3026 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
  - 3027 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 3028  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 3029 (b) If `Mask  $\neq$  GrB_NULL`,
    - 3030 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
 3031  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 3032 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
 3033  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 3034 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 3035 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 3036 4. Matrix  $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}$ .

3037 The internal matrices and masks are checked for dimension compatibility. The following conditions  
 3038 must hold:

- 3039 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 3040 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 3041 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
- 3042 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$ .
- 3043 5.  $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$ .

3044 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the dimension mismatch  
 3045 error listed above is returned.

3046 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3047 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3048 We are now ready to carry out the matrix multiplication and any additional associated operations.  
 3049 We describe this in terms of two intermediate matrices:

- 3050 •  $\tilde{\mathbf{T}}$ : The matrix holding the product of matrices  $\tilde{\mathbf{A}}$  and  $\tilde{\mathbf{B}}$ .
- 3051 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

3052 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

$$3054 \quad T_{ij} = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{B}}(:, j))} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{B}}(k, j)),$$

3055 where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring  $\mathbf{op}$ , respectively.

3056 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 3057 • If  $\mathbf{accum} = \mathbf{GrB\_NULL}$ , then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 3058 • If  $\mathbf{accum}$  is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$3059 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3060 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
3061 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 3062 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 3063 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3064 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3065 \end{aligned}$$

3066 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

3068 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
3069 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
3070 mask which acts as a “write mask”.

- 3071 • If  $\mathbf{desc}[\mathbf{GrB\_OUTP}].\mathbf{GrB\_REPLACE}$  is set, then any values in  $\mathbf{C}$  on input to this operation are  
3072 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$3073 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3074 • If  $\mathbf{desc}[\mathbf{GrB\_OUTP}].\mathbf{GrB\_REPLACE}$  is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
3075 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
3076 mask are unchanged:

$$3077 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3078 In  $\mathbf{GrB\_BLOCKING}$  mode, the method exits with return value  $\mathbf{GrB\_SUCCESS}$  and the new content  
3079 of matrix  $\mathbf{C}$  is as defined above and fully computed. In  $\mathbf{GrB\_NONBLOCKING}$  mode, the method  
3080 exits with return value  $\mathbf{GrB\_SUCCESS}$  and the new content of matrix  $\mathbf{C}$  is as defined above but  
3081 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
3082 sequence.

### 3083 4.3.2 vxm: Vector-matrix multiply

3084 Multiplies a (row) vector with a matrix on an semiring. The result is a vector.

#### 3085 C Syntax

```
3086         GrB_Info GrB_vxm(GrB_Vector          w,  
3087                           const GrB_Vector    mask,  
3088                           const GrB_BinaryOp    accum,  
3089                           const GrB_Semiring    op,  
3090                           const GrB_Vector    u,  
3091                           const GrB_Matrix     A,  
3092                           const GrB_Descriptor  desc);
```

#### 3093 Parameters

3094 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3095 that may be accumulated with the result of the vector-matrix product. On output,  
3096 this vector holds the results of the operation.

3097 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3098 stored into the output vector **w**. The mask dimensions must match those of the  
3099 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3100 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3101 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
3102 dimensions of **w**), **GrB\_NULL** should be specified.

3103 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
3104 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
3105 specified.

3106 **op** (IN) Semiring used in the vector-matrix multiply.

3107 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the  
3108 multiplication.

3109 **A** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
3110 multiplication.

3111 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
3112 should be specified. Non-default field/value pairs are listed as follows:  
3113

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.



3141 **Output** The result is written into the output vector, possibly under control of a mask.

3142 Up to four argument vectors or matrices are used in the `GrB_vxm` operation:

- 3143 1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3144 2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3145 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3146 4.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

3147 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are  
 3148 tested for domain compatibility as follows:

- 3149 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
 3150 must be from one of the pre-defined types of Table 3.2.
- 3151 2.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the semiring.
- 3152 3.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$  of the semiring.
- 3153 4.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the semiring.
- 3154 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
 3155 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$   
 3156 of the accumulation operator.

3157 Two domains are compatible with each other if values from one domain can be cast to values in  
 3158 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are  
 3159 all compatible with each other. A domain from a user-defined type is only compatible with itself.  
 3160 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the domain mismatch  
 3161 error listed above is returned.

3162 From the argument vectors and matrices, the internal matrices and mask used in the computation  
 3163 are formed ( $\leftarrow$  denotes copy):

- 3164 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3165 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 3166 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 3167 (b) If `mask`  $\neq$  `GrB_NULL`,
    - 3168 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 3169 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}\mathbf{mask}(i) = \mathbf{true})\} \rangle$ .
  - 3170 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 3171 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

3172 4. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP1}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

3173 The internal matrices and masks are checked for shape compatibility. The following conditions  
3174 must hold:

3175 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$ .

3176 2.  $\text{size}(\tilde{\mathbf{w}}) = \text{ncols}(\tilde{\mathbf{A}})$ .

3177 3.  $\text{size}(\tilde{\mathbf{u}}) = \text{nrows}(\tilde{\mathbf{A}})$ .

3178 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the dimension mismatch  
3179 error listed above is returned.

3180 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
3181 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3182 We are now ready to carry out the vector-matrix multiplication and any additional associated  
3183 operations. We describe this in terms of two intermediate vectors:

- 3184 •  $\tilde{\mathbf{t}}$ : The vector holding the product of vector  $\tilde{\mathbf{u}}^T$  and matrix  $\tilde{\mathbf{A}}$ .
- 3185 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3186 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.  
3187 The value of each of its elements is computed by

$$3188 \quad t_j = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{A}}(:, j))} (\tilde{\mathbf{u}}(k) \otimes \tilde{\mathbf{A}}(k, j)),$$

3189 where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring `op`, respectively.

3190 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3191 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 3192 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$3193 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3194 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
3195 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 3196 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 3197 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3198 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3199 \quad & \\ 3200 \end{aligned}$$

3201 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3202 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 3203 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3204 mask which acts as a “write mask”.

- 3205 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are  
 3206 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$3207 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3208 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3209 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 3210 mask are unchanged:

$$3211 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3212 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 3213 of vector  $\mathbf{w}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 3214 exits with return value GrB\_SUCCESS and the new content of vector  $\mathbf{w}$  is as defined above but  
 3215 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3216 sequence.

### 3217 4.3.3 mxv: Matrix-vector multiply

3218 Multiplies a matrix by a vector on a semiring. The result is a vector.

## 3219 C Syntax

```
3220      GrB_Info GrB_mxv(GrB_Vector      w,
3221                      const GrB_Vector mask,
3222                      const GrB_BinaryOp accum,
3223                      const GrB_Semiring op,
3224                      const GrB_Matrix A,
3225                      const GrB_Vector u,
3226                      const GrB_Descriptor desc);
```

## 3227 Parameters

3228 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
 3229 that may be accumulated with the result of the matrix-vector product. On output,  
 3230 this vector holds the results of the operation.

3231 **mask** (IN) An optional “write” mask that controls which results from this operation are  
 3232 stored into the output vector  $\mathbf{w}$ . The mask dimensions must match those of the  
 3233 vector  $\mathbf{w}$ . If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain

3234 of the `mask` vector must be of type `bool` or any of the predefined “built-in” types  
 3235 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
 3236 dimensions of `w`), `GrB_NULL` should be specified.

3237 `accum` (IN) An optional binary operator used for accumulating entries into existing `w`  
 3238 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
 3239 specified.

3240 `op` (IN) Semiring used in the vector-matrix multiply.

3241 `A` (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
 3242 multiplication.

3243 `u` (IN) The GraphBLAS vector holding the values for the right-hand vector in the  
 3244 multiplication.

3245 `desc` (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
 3246 should be specified. Non-default field/value pairs are listed as follows:

3247

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.

3248

## 3249 Return Values

3250 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
 3251 blocking mode, this indicates that the compatibility tests on di-  
 3252 mensions and domains for the input arguments passed successfully.  
 3253 Either way, output vector `w` is ready to be used in the next method  
 3254 of the sequence.

3255 `GrB_PANIC` Unknown internal error.

3256 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque  
 3257 GraphBLAS objects (input or output) is in an invalid state caused  
 3258 by a previous execution error. Call `GrB_error()` to access any error  
 3259 messages generated by the implementation.

3260 `GrB_OUT_OF_MEMORY` Not enough memory available for the operation.

3261 `GrB_UNINITIALIZED_OBJECT` One or more of the GraphBLAS objects has not been initialized by  
 3262 a call to `new` (or `dup` for matrix or vector parameters).

3263 GrB\_DIMENSION\_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

3264 GrB\_DOMAIN\_MISMATCH The domains of the various vectors/matrices are incompatible with  
3265 the corresponding domains of the semiring or accumulation opera-  
3266 tor, or the mask's domain is not compatible with **bool** (in the case  
3267 where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

## 3268 Description

3269 GrB\_mvx computes the matrix-vector product  $w = A \oplus . \otimes u$ , or, if an optional binary accumulation  
3270 operator ( $\odot$ ) is provided,  $w = w \odot (A \oplus . \otimes u)$  (where matrix  $A$  can be optionally transposed).  
3271 Logically, this operation occurs in three steps:

3272 **Setup** The internal vectors, matrices and mask used in the computation are formed and their  
3273 domains/dimensions are tested for compatibility.

3274 **Compute** The indicated computations are carried out.

3275 **Output** The result is written into the output vector, possibly under control of a mask.

3276 Up to four argument vectors or matrices are used in the GrB\_mvx operation:

- 3277 1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3278 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3279 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3280 4.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

3281 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are  
3282 tested for domain compatibility as follows:

- 3283 1. If **mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{mask})$   
3284 must be from one of the pre-defined types of Table 3.2.
- 3285 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the semiring.
- 3286 3.  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the semiring.
- 3287 4.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the semiring.
- 3288 5. If **accum** is not GrB\_NULL, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
3289 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
3290 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

$$t_i = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{A}}(i, :)) \cap \text{ind}(\tilde{\mathbf{u}})} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{u}}(k)),$$

3323 where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring **op**, respectively.

3324 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3325 • If **accum** = **GrB\_NULL**, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 3326 • If **accum** is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$3327 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3328 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 3329 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 3330 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 3331 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3332 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3333 \end{aligned}$$

3334 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

3336 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector **w**,  
 3337 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3338 mask which acts as a “write mask”.

- 3339 • If **desc[GrB\_OUTP].GrB\_REPLACE** is set, then any values in **w** on input to this operation are  
 3340 deleted and the content of the new output vector, **w**, is defined as,

$$3341 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3342 • If **desc[GrB\_OUTP].GrB\_REPLACE** is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3343 copied into the result vector, **w**, and elements of **w** that fall outside the set indicated by the  
 3344 mask are unchanged:

$$3345 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3346 In **GrB\_BLOCKING** mode, the method exits with return value **GrB\_SUCCESS** and the new content  
 3347 of vector **w** is as defined above and fully computed. In **GrB\_NONBLOCKING** mode, the method  
 3348 exits with return value **GrB\_SUCCESS** and the new content of vector **w** is as defined above but  
 3349 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3350 sequence.

#### 3351 4.3.4 eWiseMult: Element-wise multiplication

3352 **Note:** The difference between **eWiseAdd** and **eWiseMult** is not about the element-wise operation  
 3353 but how the index sets are treated. **eWiseAdd** returns an object whose indices are the “union” of  
 3354 the indices of the inputs whereas **eWiseMult** returns an object whose indices are the “intersection”  
 3355 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on  
 3356 the set of values from the resulting index set.

#### 4.3.4.1 eWiseMult: Vector variant

Perform element-wise (general) multiplication on the intersection of elements of two vectors, producing a third vector as result.

#### C Syntax

```
GrB_Info GrB_eWiseMult(GrB_Vector      w,
                        const GrB_Vector mask,
                        const GrB_BinaryOp accum,
                        const GrB_Semiring op,
                        const GrB_Vector u,
                        const GrB_Vector v,
                        const GrB_Descriptor desc);

GrB_Info GrB_eWiseMult(GrB_Vector      w,
                        const GrB_Vector mask,
                        const GrB_BinaryOp accum,
                        const GrB_Monoid op,
                        const GrB_Vector u,
                        const GrB_Vector v,
                        const GrB_Descriptor desc);

GrB_Info GrB_eWiseMult(GrB_Vector      w,
                        const GrB_Vector mask,
                        const GrB_BinaryOp accum,
                        const GrB_BinaryOp op,
                        const GrB_Vector u,
                        const GrB_Vector v,
                        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 element-wise 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) 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.

3425 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
 3426 a call to `new` (or `dup` for vector parameters).

3427 GrB\_DIMENSION\_MISMATCH Mask or vector dimensions are incompatible.

3428 GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with the cor-  
 3429 responding domains of the binary operator (`op`) or accumulation  
 3430 operator, or the mask's domain is not compatible with `bool` (in the  
 3431 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3432 Description

3433 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS vectors:  
 3434  $\mathbf{w} = \mathbf{u} \otimes \mathbf{v}$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $\mathbf{w} = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$ .  
 3435 Logically, this operation occurs in three steps:

3436 **Setup** The internal vectors and mask used in the computation are formed and their domains  
 3437 and dimensions are tested for compatibility.

3438 **Compute** The indicated computations are carried out.

3439 **Output** The result is written into the output vector, possibly under control of a mask.

3440 Up to four argument vectors are used in the `GrB_eWiseMult` operation:

- 3441 1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3442 2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3443 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3444 4.  $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

3445 The argument vectors, the “product” operator (`op`), and the accumulation operator (if provided)  
 3446 are tested for domain compatibility as follows:

- 3447 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
 3448 must be from one of the pre-defined types of Table 3.2.
- 3449 2.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$ .
- 3450 3.  $\mathbf{D}(\mathbf{v})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$ .
- 3451 4.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$ .
- 3452 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
 3453 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of  
 3454 the accumulation operator.

3455 Two domains are compatible with each other if values from one domain can be cast to values in  
 3456 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 3457 compatible with each other. A domain from a user-defined type is only compatible with itself. If any  
 3458 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch  
 3459 error listed above is returned.

3460 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
 3461 denotes copy):

- 3462 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3463 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 3464 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .
  - 3465 (b) If `mask  $\neq$  GrB_NULL`,
    - 3466 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
    - 3467 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 3468 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 3469 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 3470 4. Vector  $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$ .

3471 The internal vectors and mask are checked for dimension compatibility. The following conditions  
 3472 must hold:

- 3473 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}}) = \text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{v}})$ .

3474 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension  
 3475 mismatch error listed above is returned.

3476 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3477 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3478 We are now ready to carry out the element-wise “product” and any additional associated operations.  
 3479 We describe this in terms of two intermediate vectors:

- 3480 •  $\tilde{\mathbf{t}}$ : The vector holding the element-wise “product” of  $\tilde{\mathbf{u}}$  and vector  $\tilde{\mathbf{v}}$ .
- 3481 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3482 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  
 3483 value of each of its elements is computed by:

$$3484 \quad t_i = (\tilde{\mathbf{u}}(i) \otimes \tilde{\mathbf{v}}(i)), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}}))$$

3485 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

3486 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .

3487 • If  $\text{accum}$  is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

3488 
$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3489 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
3490 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

3491 
$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

3492

3493 
$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3494

3495 
$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3496 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3497 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
3498 using what is called a *standard vector mask and replace*. This is carried out under control of the  
3499 mask which acts as a “write mask”.

3500 • If  $\text{desc}[\text{GrB\_OUTP}].\text{GrB\_REPLACE}$  is set, then any values in  $\mathbf{w}$  on input to this operation are  
3501 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

3502 
$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3503 • If  $\text{desc}[\text{GrB\_OUTP}].\text{GrB\_REPLACE}$  is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
3504 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
3505 mask are unchanged:

3506 
$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3507 In  $\text{GrB\_BLOCKING}$  mode, the method exits with return value  $\text{GrB\_SUCCESS}$  and the new content  
3508 of vector  $\mathbf{w}$  is as defined above and fully computed. In  $\text{GrB\_NONBLOCKING}$  mode, the method  
3509 exits with return value  $\text{GrB\_SUCCESS}$  and the new content of vector  $\mathbf{w}$  is as defined above but  
3510 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
3511 sequence.

#### 3512 4.3.4.2 eWiseMult: Matrix variant

3513 Perform element-wise (general) multiplication on the intersection of elements of two matrices, pro-  
3514 ducing a third matrix as result.

## 3515 C Syntax

```

3516         GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3517                                const GrB_Matrix Mask,
3518                                const GrB_BinaryOp accum,
3519                                const GrB_Semiring op,
3520                                const GrB_Matrix A,
3521                                const GrB_Matrix B,
3522                                const GrB_Descriptor desc);
3523
3524         GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3525                                const GrB_Matrix Mask,
3526                                const GrB_BinaryOp accum,
3527                                const GrB_Monoid op,
3528                                const GrB_Matrix A,
3529                                const GrB_Matrix B,
3530                                const GrB_Descriptor desc);
3531
3532         GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3533                                const GrB_Matrix Mask,
3534                                const GrB_BinaryOp accum,
3535                                const GrB_BinaryOp op,
3536                                const GrB_Matrix A,
3537                                const GrB_Matrix B,
3538                                const GrB_Descriptor desc);

```

## 3539 Parameters

3540 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
3541 that may be accumulated with the result of the element-wise operation. On output,  
3542 the matrix holds the results of the operation.

3543 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
3544 stored into the output matrix C. The mask dimensions must match those of the  
3545 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
3546 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types  
3547 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
3548 dimensions of C), GrB\_NULL should be specified.

3549 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
3550 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
3551 specified.

3552 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”  
3553 operation. Depending on which type is passed, the following defines the binary  
3554 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

3555 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .  
 3556 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-  
 3557 nored.  
 3558 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  
 3559 is ignored.

3560 A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
 3561 operation.

3562 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
 3563 operation.

3564 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 3565 should be specified. Non-default field/value pairs are listed as follows:  
 3566

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.

## 3568 Return Values

3569 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 3570 blocking mode, this indicates that the compatibility tests on di-  
 3571 mensions and domains for the input arguments passed successfully.  
 3572 Either way, output matrix C is ready to be used in the next method  
 3573 of the sequence.

3574 GrB\_PANIC Unknown internal error.

3575 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 3576 GraphBLAS objects (input or output) is in an invalid state caused  
 3577 by a previous execution error. Call GrB\_error() to access any error  
 3578 messages generated by the implementation.

3579 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

3580 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
 3581 a call to new (or Matrix\_dup for matrix parameters).

3582 GrB\_DIMENSION\_MISMATCH Mask and/or matrix dimensions are incompatible.

3583 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
 3584 corresponding domains of the binary operator ( $\otimes$ ) or accumulation  
 3585 operator, or the mask's domain is not compatible with `bool` (in the  
 3586 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3587 Description

3588 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS matrices:  
 3589  $C = A \otimes B$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot (A \otimes B)$ .  
 3590 Logically, this operation occurs in three steps:

3591 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 3592 and dimensions are tested for compatibility.

3593 **Compute** The indicated computations are carried out.

3594 **Output** The result is written into the output matrix, possibly under control of a mask.

3595 Up to four argument matrices are used in the `GrB_eWiseMult` operation:

- 3596 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3597 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)
- 3598 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3599 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3600 The argument matrices, the “product” operator ( $\otimes$ ), and the accumulation operator (if provided)  
 3601 are tested for domain compatibility as follows:

- 3602 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 3603 must be from one of the pre-defined types of Table 3.2.
- 3604 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\otimes)$ .
- 3605 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\otimes)$ .
- 3606 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\otimes)$ .
- 3607 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 3608 of the accumulation operator and  $\mathbf{D}_{out}(\otimes)$  of  $\otimes$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
 3609 the accumulation operator.

3610 Two domains are compatible with each other if values from one domain can be cast to values in  
 3611 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 3612 compatible with each other. A domain from a user-defined type is only compatible with itself. If any

3613 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch  
 3614 error listed above is returned.

3615 From the argument matrices, the internal matrices and mask used in the computation are formed  
 3616 ( $\leftarrow$  denotes copy):

- 3617 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 3618 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
  - 3619 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 3620  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 3621 (b) If `Mask  $\neq$  GrB_NULL`,
    - 3622 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
 3623  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 3624 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
 3625  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\text{bool})\mathbf{Mask}(i, j) = \text{true}\} \rangle$ .
  - 3626 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 3627 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 3628 4. Matrix  $\tilde{\mathbf{B}} \leftarrow \text{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}$ .

3629 The internal matrices and masks are checked for dimension compatibility. The following conditions  
 3630 must hold:

- 3631 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$ .
- 3632 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$ .

3633 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension  
 3634 mismatch error listed above is returned.

3635 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3636 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3637 We are now ready to carry out the element-wise “product” and any additional associated operations.  
 3638 We describe this in terms of two intermediate matrices:

- 3639 •  $\tilde{\mathbf{T}}$ : The matrix holding the element-wise product of  $\tilde{\mathbf{A}}$  and  $\tilde{\mathbf{B}}$ .
- 3640 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

3641 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\} \rangle$   
 3642 is created. The value of each of its elements is computed by

$$3643 \quad T_{ij} = (\tilde{\mathbf{A}}(i, j) \otimes \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})$$

3644 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:



3645 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .

3646 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$3647 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3648 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
3649 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$3650 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$3651 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$3652 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3653 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3656 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
3657 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
3658 mask which acts as a “write mask”.

3659 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
3660 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$3661 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3662 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
3663 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
3664 mask are unchanged:

$$3665 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3666 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
3667 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
3668 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
3669 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
3670 sequence.

### 3671 4.3.5 eWiseAdd: Element-wise addition

3672 **Note:** The difference between `eWiseAdd` and `eWiseMult` is not about the element-wise operation  
3673 but how the index sets are treated. `eWiseAdd` returns an object whose indices are the “union” of  
3674 the indices of the inputs whereas `eWiseMult` returns an object whose indices are the “intersection”  
3675 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on  
3676 the set of values from the resulting index set.

#### 3677 4.3.5.1 eWiseAdd: Vector variant

3678 Perform element-wise (general) addition on the elements of two vectors, producing a third vector  
3679 as result.

#### 3680 C Syntax

```
3681     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3682                          const GrB_Vector mask,  
3683                          const GrB_BinaryOp accum,  
3684                          const GrB_Semiring op,  
3685                          const GrB_Vector u,  
3686                          const GrB_Vector v,  
3687                          const GrB_Descriptor desc);  
3688  
3689     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3690                          const GrB_Vector mask,  
3691                          const GrB_BinaryOp accum,  
3692                          const GrB_Monoid op,  
3693                          const GrB_Vector u,  
3694                          const GrB_Vector v,  
3695                          const GrB_Descriptor desc);  
3696  
3697     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3698                          const GrB_Vector mask,  
3699                          const GrB_BinaryOp accum,  
3700                          const GrB_BinaryOp op,  
3701                          const GrB_Vector u,  
3702                          const GrB_Vector v,  
3703                          const GrB_Descriptor desc);
```

#### 3704 Parameters

3705 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3706 that may be accumulated with the result of the element-wise operation. On output,  
3707 this vector holds the results of the operation.

3708 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3709 stored into the output vector **w**. The mask dimensions must match those of the  
3710 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3711 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3712 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
3713 dimensions of **w**), **GrB\_NULL** should be specified.

3714 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**

3715 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
3716 specified.

3717 op (IN) The semiring, monoid, or binary operator used in the element-wise “sum”  
3718 operation. Depending on which type is passed, the following defines the binary  
3719 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$ , used:

3720 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

3721 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-  
3722 nored.

3723 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-  
3724 nary op and additive identity are ignored.

3725 u (IN) The GraphBLAS vector holding the values for the left-hand vector in the  
3726 operation.

3727 v (IN) The GraphBLAS vector holding the values for the right-hand vector in the  
3728 operation.

3729 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
3730 should be specified. Non-default field/value pairs are listed as follows:  
3731

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.

## 3733 Return Values

3734 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
3735 blocking mode, this indicates that the compatibility tests on di-  
3736 mensions and domains for the input arguments passed successfully.  
3737 Either way, output vector w is ready to be used in the next method  
3738 of the sequence.

3739 GrB\_PANIC Unknown internal error.

3740 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
3741 GraphBLAS objects (input or output) is in an invalid state caused  
3742 by a previous execution error. Call GrB\_error() to access any error  
3743 messages generated by the implementation.

3744 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

3745 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
3746 a call to `new` (or `dup` for vector parameters).

3747 GrB\_DIMENSION\_MISMATCH Mask or vector dimensions are incompatible.

3748 GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with the cor-  
3749 responding domains of the binary operator (`op`) or accumulation  
3750 operator, or the mask's domain is not compatible with `bool` (in the  
3751 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3752 Description

3753 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS vectors:  $w =$   
3754  $u \oplus v$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $w = w \odot (u \oplus v)$ . Logically,  
3755 this operation occurs in three steps:

3756 **Setup** The internal vectors and mask used in the computation are formed and their domains  
3757 and dimensions are tested for compatibility.

3758 **Compute** The indicated computations are carried out.

3759 **Output** The result is written into the output vector, possibly under control of a mask.

3760 Up to four argument vectors are used in the `GrB_eWiseAdd` operation:

- 3761 1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3762 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3763 3.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$
- 3764 4.  $v = \langle \mathbf{D}(v), \mathbf{size}(v), \mathbf{L}(v) = \{(i, v_i)\} \rangle$

3765 The argument vectors, the “sum” operator (`op`), and the accumulation operator (if provided) are  
3766 tested for domain compatibility as follows:

- 3767 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
3768 must be from one of the pre-defined types of Table 3.2.
- 3769 2.  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$ .
- 3770 3.  $\mathbf{D}(v)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$ .
- 3771 4.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3772 5.  $\mathbf{D}(u)$  and  $\mathbf{D}(v)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3773 6. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
3774 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
3775 the accumulation operator.

3776 Two domains are compatible with each other if values from one domain can be cast to values in  
 3777 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 3778 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 3779 any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the domain mismatch  
 3780 error listed above is returned.

3781 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
 3782 denotes copy):

- 3783 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3784 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 3785 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .
  - 3786 (b) If `mask  $\neq$  GrB_NULL`,
    - 3787 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
    - 3788 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 3789 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 3790 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 3791 4. Vector  $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$ .

3792 The internal vectors and mask are checked for dimension compatibility. The following conditions  
 3793 must hold:

- 3794 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}}) = \text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{v}})$ .

3795 If any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the dimension  
 3796 mismatch error listed above is returned.

3797 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3798 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3799 We are now ready to carry out the element-wise “sum” and any additional associated operations.  
 3800 We describe this in terms of two intermediate vectors:

- 3801 •  $\tilde{\mathbf{t}}$ : The vector holding the element-wise “sum” of  $\tilde{\mathbf{u}}$  and vector  $\tilde{\mathbf{v}}$ .
- 3802 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3803 The intermediate vector  $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, t_i) : \text{ind}(\tilde{\mathbf{u}}) \cup \text{ind}(\tilde{\mathbf{v}}) \neq \emptyset\} \rangle$  is created. The  
 3804 value of each of its elements is computed by:

$$\begin{aligned}
 3805 \quad t_i &= (\tilde{\mathbf{u}}(i) \oplus \tilde{\mathbf{v}}(i)), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}})) \\
 3806 \\
 3807 \quad t_i &= \tilde{\mathbf{u}}(i), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) - (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}})))
 \end{aligned}$$

3808  
3809

$$t_i = \tilde{\mathbf{v}}(i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{v}}) - (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}})))$$

3810 where the difference operator in the previous expressions refers to set difference.

3811 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3812 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 3813 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3815 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
3816 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 3817 z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 3818 z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3819 z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3820 \end{aligned}$$

3822 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3823 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
3824 using what is called a *standard vector mask and replace*. This is carried out under control of the  
3825 mask which acts as a “write mask”.

- 3826 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
3827 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3829 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
3830 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
3831 mask are unchanged:

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3833 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
3834 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
3835 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but  
3836 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
3837 sequence.

#### 3838 4.3.5.2 eWiseAdd: Matrix variant

3839 Perform element-wise (general) addition on the elements of two matrices, producing a third matrix  
3840 as result.

## 3841 C Syntax

```

3842     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3843                           const GrB_Matrix Mask,
3844                           const GrB_BinaryOp accum,
3845                           const GrB_Semiring op,
3846                           const GrB_Matrix A,
3847                           const GrB_Matrix B,
3848                           const GrB_Descriptor desc);
3849
3850     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3851                           const GrB_Matrix Mask,
3852                           const GrB_BinaryOp accum,
3853                           const GrB_Monoid op,
3854                           const GrB_Matrix A,
3855                           const GrB_Matrix B,
3856                           const GrB_Descriptor desc);
3857
3858     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3859                           const GrB_Matrix Mask,
3860                           const GrB_BinaryOp accum,
3861                           const GrB_BinaryOp op,
3862                           const GrB_Matrix A,
3863                           const GrB_Matrix B,
3864                           const GrB_Descriptor desc);

```

## 3865 Parameters

3866 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
3867 that may be accumulated with the result of the element-wise operation. On output,  
3868 the matrix holds the results of the operation.

3869 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
3870 stored into the output matrix C. The mask dimensions must match those of the  
3871 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
3872 of the **Mask** matrix must be of type `bool` or any of the predefined “built-in” types  
3873 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
3874 dimensions of C), `GrB_NULL` should be specified.

3875 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
3876 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
3877 specified.

3878 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”  
3879 operation. Depending on which type is passed, the following defines the binary  
3880 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$ , used:

3881  
3882  
3883  
3884  
3885  
  
3886  
3887  
  
3888  
3889  
  
3890  
3891  
3892  
  
3893  
  
  
  
3894  
  
3895  
3896  
3897  
3898  
3899  
  
3900  
  
3901  
3902  
3903  
3904  
  
3905  
  
3906  
3907  
  
3908

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.

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.



3909 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
 3910 corresponding domains of the binary operator ( $\text{op}$ ) or accumulation  
 3911 operator, or the mask's domain is not compatible with `bool` (in the  
 3912 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3913 Description

3914 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS matrices:  
 3915  $C = A \oplus B$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot (A \oplus B)$ .  
 3916 Logically, this operation occurs in three steps:

3917 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 3918 and dimensions are tested for compatibility.

3919 **Compute** The indicated computations are carried out.

3920 **Output** The result is written into the output matrix, possibly under control of a mask.

3921 Up to four argument matrices are used in the `GrB_eWiseAdd` operation:

- 3922 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3923 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)
- 3924 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3925 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3926 The argument matrices, the “sum” operator ( $\text{op}$ ), and the accumulation operator (if provided) are  
 3927 tested for domain compatibility as follows:

- 3928 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 3929 must be from one of the pre-defined types of Table 3.2.
- 3930 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$ .
- 3931 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$ .
- 3932 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3933 5.  $\mathbf{D}(A)$  and  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3934 6. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 3935 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of  $\text{op}$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
 3936 the accumulation operator.

3937 Two domains are compatible with each other if values from one domain can be cast to values in  
 3938 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 3939 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 3940 any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the domain mismatch  
 3941 error listed above is returned.

3942 From the argument matrices, the internal matrices and mask used in the computation are formed  
 3943 ( $\leftarrow$  denotes copy):

- 3944 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 3945 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
  - 3946 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 3947  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 3948 (b) If `Mask  $\neq$  GrB_NULL`,
    - 3949 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
 3950  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 3951 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
 3952  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 3953 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 3954 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 3955 4. Matrix  $\tilde{\mathbf{B}} \leftarrow \text{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}$ .

3956 The internal matrices and masks are checked for dimension compatibility. The following conditions  
 3957 must hold:

- 3958 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$ .
- 3959 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$ .

3960 If any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the dimension  
 3961 mismatch error listed above is returned.

3962 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3963 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3964 We are now ready to carry out the element-wise “sum” and any additional associated operations.  
 3965 We describe this in terms of two intermediate matrices:

- 3966 •  $\tilde{\mathbf{T}}$ : The matrix holding the element-wise sum of  $\tilde{\mathbf{A}}$  and  $\tilde{\mathbf{B}}$ .
- 3967 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

3968 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cup \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\}$   
 3969 is created. The value of each of its elements is computed by

$$\begin{aligned} 3970 \quad T_{ij} &= (\tilde{\mathbf{A}}(i, j) \oplus \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \\ 3971 \quad T_{ij} &= \tilde{\mathbf{A}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{A}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}))) \\ 3972 \quad T_{ij} &= \tilde{\mathbf{B}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{B}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}))) \end{aligned}$$

3975 where the difference operator in the previous expressions refers to set difference.

3976 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 3977 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 3978 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$3979 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3980 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 3981 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 3982 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 3983 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3984 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \end{aligned}$$

3987 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3988 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 3989 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 3990 mask which acts as a “write mask”.

- 3991 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
 3992 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$3993 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3994 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 3995 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 3996 mask are unchanged:

$$3997 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3998 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
 3999 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
 4000 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
 4001 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 4002 sequence.

### 4003 4.3.6 extract: Selecting sub-graphs

4004 Extract a subset of a matrix or vector.

#### 4005 4.3.6.1 extract: Standard vector variant

4006 Extract a sub-vector from a larger vector as specified by a set of indices. The result is a vector  
4007 whose size is equal to the number of indices.

### 4008 C Syntax

```
4009         GrB_Info GrB_extract(GrB_Vector          w,  
4010                             const GrB_Vector    mask,  
4011                             const GrB_BinaryOp   accum,  
4012                             const GrB_Vector    u,  
4013                             const GrB_Index     *indices,  
4014                             GrB_Index          nindices,  
4015                             const GrB_Descriptor desc);
```

### 4016 Parameters

4017 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
4018 that may be accumulated with the result of the extract operation. On output, this  
4019 vector holds the results of the operation.

4020 **mask** (IN) An optional “write” mask that controls which results from this operation are  
4021 stored into the output vector **w**. The mask dimensions must match those of the  
4022 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4023 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
4024 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4025 dimensions of **w**), **GrB\_NULL** should be specified.

4026 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
4027 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4028 specified.

4029 **u** (IN) The GraphBLAS vector from which the subset is extracted.

4030 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations of  
4031 elements from **u** that are extracted. If all elements of **u** are to be extracted in order  
4032 from 0 to **nindices** – 1, then **GrB\_ALL** should be specified. Regardless of execution  
4033 mode and return value, this array may be manipulated by the caller after this  
4034 operation returns without affecting any deferred computations for this operation.

4035 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(w)**.

4036 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 4037 should be specified. Non-default field/value pairs are listed as follows:  
 4038

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> .

## 4040 Return Values

4041 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 4042 blocking mode, this indicates that the compatibility tests on  
 4043 dimensions and domains for the input arguments passed suc-  
 4044 cessfully. Either way, output vector **w** is ready to be used in the  
 4045 next method of the sequence.

4046 GrB\_PANIC Unknown internal error.

4047 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
 4048 opaque GraphBLAS objects (input or output) is in an invalid  
 4049 state caused by a previous execution error. Call GrB\_error() to  
 4050 access any error messages generated by the implementation.

4051 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

4052 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
 4053 by a call to **new** (or **dup** for vector parameters).

4054 GrB\_INDEX\_OUT\_OF\_BOUNDS A value in **indices** is greater than or equal to **size(u)**. In non-  
 4055 blocking mode, this error can be deferred.

4056 GrB\_DIMENSION\_MISMATCH **mask** and **w** dimensions are incompatible, or **nindices**  $\neq$  **size(w)**.

4057 GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with each  
 4058 other or the corresponding domains of the accumulation oper-  
 4059 ator, or the mask's domain is not compatible with **bool** (in the  
 4060 case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

4061 GrB\_NULL\_POINTER Argument **row\_indices** is a NULL pointer.

## 4062 Description

4063 This variant of GrB\_extract computes the result of extracting a subset of locations from a Graph-  
 4064 BLAS vector in a specific order:  $\mathbf{w} = \mathbf{u}(\mathbf{indices})$ ; or, if an optional binary accumulation operator

4065  $(\odot)$  is provided,  $w = w \odot u(\text{indices})$ . More explicitly:

$$4066 \quad \begin{aligned} w(i) &= u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ w(i) &= w(i) \odot u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices} \end{aligned}$$

4067 Logically, this operation occurs in three steps:

4068     **Setup** The internal vectors and mask used in the computation are formed and their domains  
4069             and dimensions are tested for compatibility.

4070     **Compute** The indicated computations are carried out.

4071     **Output** The result is written into the output vector, possibly under control of a mask.

4072 Up to three argument vectors are used in this GrB\_extract operation:

- 4073     1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4074     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4075     3.  $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

4076 The argument vectors and the accumulation operator (if provided) are tested for domain compati-  
4077 bility as follows:

- 4078     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
4079         must be from one of the pre-defined types of Table 3.2.
- 4080     2.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}(u)$ .
- 4081     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
4082         of the accumulation operator and  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
4083         mulation operator.

4084 Two domains are compatible with each other if values from one domain can be cast to values in  
4085 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4086 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4087 any compatibility rule above is violated, execution of GrB\_extract ends and the domain mismatch  
4088 error listed above is returned.

4089 From the arguments, the internal vectors, mask, and index array used in the computation are  
4090 formed ( $\leftarrow$  denotes copy):

- 4091     1. Vector  $\tilde{w} \leftarrow w$ .
- 4092     2. One-dimensional mask,  $\tilde{m}$ , is computed from argument `mask` as follows:  
4093         (a) If `mask = GrB_NULL`, then  $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$ .

- 4094 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,  
 4095 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,  
 4096 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .  
 4097 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .
- 4098 3. Vector  $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 4099 4. The internal index array,  $\widetilde{\mathbf{I}}$ , is computed from argument indices as follows:
- 4100 (a) If  $\text{indices} = \text{GrB\_ALL}$ , then  $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$ .  
 4101 (b) Otherwise,  $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$ .

4102 The internal vectors and mask are checked for dimension compatibility. The following conditions  
 4103 must hold:

- 4104 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$   
 4105 2.  $\text{nindices} = \text{size}(\widetilde{\mathbf{w}})$ .

4106 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
 4107 match error listed above is returned.

4108 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 4109 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4110 We are now ready to carry out the extract and any additional associated operations. We describe  
 4111 this in terms of two intermediate vectors:

- 4112 •  $\widetilde{\mathbf{t}}$ : The vector holding the extraction from  $\widetilde{\mathbf{u}}$  in their destination locations relative to  $\widetilde{\mathbf{w}}$ .
- 4113 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4114 The intermediate vector,  $\widetilde{\mathbf{t}}$ , is created as follows:

$$4115 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{w}}), \{(i, \widetilde{\mathbf{u}}[\widetilde{\mathbf{I}}[i]]) \mid \forall i, 0 \leq i < \text{nindices} : \widetilde{\mathbf{I}}[i] \in \text{ind}(\widetilde{\mathbf{u}})\} \rangle.$$

4116 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  
 4117 `GrB_extract` ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING`  
 4118 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
 4119 result vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4120 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 4121 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$ .
- 4122 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{z}}$  is defined as

$$4123 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

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.

## 4190 Return Values

4191	<b>GrB_SUCCESS</b>	In blocking mode, the operation completed successfully. In non-
4192		blocking mode, this indicates that the compatibility tests on
4193		dimensions and domains for the input arguments passed suc-
4194		cessfully. Either way, output matrix C is ready to be used in the
4195		next method of the sequence.
4196	<b>GrB_PANIC</b>	Unknown internal error.
4197	<b>GrB_INVALID_OBJECT</b>	This is returned in any execution mode whenever one of the
4198		opaque GraphBLAS objects (input or output) is in an invalid
4199		state caused by a previous execution error. Call <code>GrB_error()</code> to
4200		access any error messages generated by the implementation.
4201	<b>GrB_OUT_OF_MEMORY</b>	Not enough memory available for the operation.
4202	<b>GrB_UNINITIALIZED_OBJECT</b>	One or more of the GraphBLAS objects has not been initialized
4203		by a call to <code>new</code> (or <code>Matrix_dup</code> for matrix parameters).
4204	<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
4205		a value in <code>col_indices</code> is greater than or equal to <code>ncols(A)</code> . In
4206		non-blocking mode, this error can be deferred.
4207	<b>GrB_DIMENSION_MISMATCH</b>	Mask and C dimensions are incompatible, <code>nrows</code> $\neq$ <code>nrows(C)</code> , or
4208		<code>ncols</code> $\neq$ <code>ncols(C)</code> .
4209	<b>GrB_DOMAIN_MISMATCH</b>	The domains of the various matrices are incompatible with each
4210		other or the corresponding domains of the accumulation oper-
4211		ator, or the mask's domain is not compatible with <code>bool</code> (in the
4212		case where <code>desc[GrB_MASK].GrB_STRUCTURE</code> is not set).
4213	<b>GrB_NULL_POINTER</b>	Either argument <code>row_indices</code> is a NULL pointer, argument <code>col_indices</code>
4214		is a NULL pointer, or both.

## 4215 Description

4216 This variant of `GrB_extract` computes the result of extracting a subset of locations from specified  
 4217 rows and columns of a GraphBLAS matrix in a specific order:  $C = A(\text{row\_indices}, \text{col\_indices})$ ; or,  
 4218 if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot A(\text{row\_indices}, \text{col\_indices})$ .  
 4219 More explicitly (not accounting for an optional transpose of A):

$$\begin{aligned}
 &C(i, j) = A(\text{row\_indices}[i], \text{col\_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\
 &C(i, j) = C(i, j) \odot A(\text{row\_indices}[i], \text{col\_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}
 \end{aligned}$$

4221 Logically, this operation occurs in three steps:

4222 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 4223 and dimensions are tested for compatibility.

4224 **Compute** The indicated computations are carried out.

4225 **Output** The result is written into the output matrix, possibly under control of a mask.

4226 Up to three argument matrices are used in the `GrB_extract` operation:

- 4227 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4228 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)
- 4229 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4230 The argument matrices and the accumulation operator (if provided) are tested for domain compat-  
4231 ibility as follows:

- 4232 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
4233 must be from one of the pre-defined types of Table 3.2.
- 4234 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$ .
- 4235 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
4236 of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
4237 mulation operator.

4238 Two domains are compatible with each other if values from one domain can be cast to values in  
4239 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4240 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4241 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch  
4242 error listed above is returned.

4243 From the arguments, the internal matrices, `mask`, and index arrays used in the computation are  
4244 formed ( $\leftarrow$  denotes copy):

- 4245 1. Matrix  $\tilde{C} \leftarrow C$ .
- 4246 2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument `Mask` as follows:
  - 4247 (a) If `Mask` = `GrB_NULL`, then  $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$   
4248  $j < \mathbf{ncols}(C)\} \rangle$ .
  - 4249 (b) If `Mask`  $\neq$  `GrB_NULL`,
    - 4250 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
4251  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
    - 4252 ii. Otherwise,  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
4253  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
  - 4254 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{M} \leftarrow \neg \tilde{M}$ .
- 4255 3. Matrix  $\tilde{A} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? A^T : A$ .

- 4256 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
- 4257 (a) If `row_indices` = `GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$ .
- 4258 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{row\_indices}[i], \forall i : 0 \leq i < \text{nrows}$ .
- 4259 5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:
- 4260 (a) If `col_indices` = `GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \text{ncols}$ .
- 4261 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \text{col\_indices}[j], \forall j : 0 \leq j < \text{ncols}$ .

4262 The internal matrices and mask are checked for dimension compatibility. The following conditions  
4263 must hold:

- 4264 1.  $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}(\tilde{\mathbf{M}})$ .
- 4265 2.  $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}(\tilde{\mathbf{M}})$ .
- 4266 3.  $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}$ .
- 4267 4.  $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}$ .

4268 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
4269 match error listed above is returned.

4270 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4271 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4272 We are now ready to carry out the extract and any additional associated operations. We describe  
4273 this in terms of two intermediate matrices:

- 4274 •  $\tilde{\mathbf{T}}$ : The matrix holding the extraction from  $\tilde{\mathbf{A}}$ .
- 4275 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

4276 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

4277 
$$\tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \{ (i, j, \tilde{\mathbf{A}}(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j])) \mid \forall (i, j), 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} : (\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j]) \in \text{ind}(\tilde{\mathbf{A}}) \} \rangle.$$

4278 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  $\tilde{\mathbf{J}}$   
4279 array is not in the range  $[0, \text{ncols}(\tilde{\mathbf{A}}))$ , the execution of `GrB_extract` ends and the index out-of-  
4280 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred  
4281 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from  
4282 this point forward in the sequence.

4283 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 4284 • If `accum` = `GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .

4285 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$4286 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4287 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
4288 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$4289 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$4290 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$4292 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4294 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4295 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
4296 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
4297 mask which acts as a “write mask”.

4298 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
4299 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$4300 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4301 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
4302 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
4303 mask are unchanged:

$$4304 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4305 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
4306 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
4307 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
4308 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
4309 sequence.

#### 4310 4.3.6.3 extract: Column (and row) variant

4311 Extract from one column of a matrix into a vector. Note that with the transpose descriptor for the  
4312 source matrix, elements of an arbitrary row of the matrix can be extracted with this function as  
4313 well.

## 4314 C Syntax

```

4315         GrB_Info GrB_extract(GrB_Vector      w,
4316                             const GrB_Vector  mask,
4317                             const GrB_BinaryOp accum,
4318                             const GrB_Matrix  A,
4319                             const GrB_Index   *row_indices,
4320                             GrB_Index         nrows,
4321                             GrB_Index         col_index,
4322                             const GrB_Descriptor desc);

```

## 4323 Parameters

4324        **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
4325        that may be accumulated with the result of the extract operation. On output, this  
4326        vector holds the results of the operation.

4327        **mask** (IN) An optional “write” mask that controls which results from this operation are  
4328        stored into the output vector **w**. The mask dimensions must match those of the  
4329        vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4330        of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
4331        in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4332        dimensions of **w**), **GrB\_NULL** should be specified.

4333        **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
4334        entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4335        specified.

4336        **A** (IN) The GraphBLAS matrix from which the column subset is extracted.

4337        **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations  
4338        within the specified column of **A** from which elements are extracted. If elements in  
4339        all rows of **A** are to be extracted in order, **GrB\_ALL** should be specified. Regardless  
4340        of execution mode and return value, this array may be manipulated by the caller  
4341        after this operation returns without affecting any deferred computations for this  
4342        operation.

4343        **nrows** (IN) The number of indices in the **row\_indices** array. Must be equal to **size(w)**.

4344        **col\_index** (IN) The index of the column of **A** from which to extract values. It must be in the  
4345        range  $[0, \mathbf{ncols}(A))$ .

4346        **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
4347        should be specified. Non-default field/value pairs are listed as follows:

4348

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

## Return Values

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

**GrB\_PANIC** Unknown internal error.

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

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

**GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by a call to **new** (or **dup** for vector or matrix parameters).

**GrB\_INVALID\_INDEX** **col\_index** is outside the allowable range (i.e., greater than **ncols(A)**).

**GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **row\_indices** is greater than or equal to **nrows(A)**. In non-blocking mode, this error can be deferred.

**GrB\_DIMENSION\_MISMATCH** **mask** and **w** dimensions are incompatible, or **nrows**  $\neq$  **size(w)**.

**GrB\_DOMAIN\_MISMATCH** The domains of the vector or matrix are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with **bool** (in the case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

**GrB\_NULL\_POINTER** Argument **row\_indices** is a NULL pointer.

## Description

This variant of **GrB\_extract** computes the result of extracting a subset of locations (in a specific order) from a specified column of a GraphBLAS matrix: **w** = **A(:, col\_index)(row\_indices)**; or, if

4376 an optional binary accumulation operator ( $\odot$ ) is provided,  $w = w \odot A(:, \text{col\_index})(\text{row\_indices})$ .  
 4377 More explicitly:

$$4378 \quad \begin{aligned} w(i) &= A(\text{row\_indices}[i], \text{col\_index}) \quad \forall i : 0 \leq i < \text{nrows}, \quad \text{or} \\ w(i) &= w(i) \odot A(\text{row\_indices}[i], \text{col\_index}) \quad \forall i : 0 \leq i < \text{nrows} \end{aligned}$$

4379 Logically, this operation occurs in three steps:

4380     **Setup** The internal matrices, vectors, and mask used in the computation are formed and their  
 4381 domains and dimensions are tested for compatibility.

4382     **Compute** The indicated computations are carried out.

4383     **Output** The result is written into the output vector, possibly under control of a mask.

4384 Up to three argument vectors and matrices are used in this GrB\_extract operation:

- 4385     1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4386     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4387     3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4388 The argument vectors, matrix and the accumulation operator (if provided) are tested for domain  
 4389 compatibility as follows:

- 4390     1. If **mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{mask})$   
 4391 must be from one of the pre-defined types of Table 3.2.
- 4392     2.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}(A)$ .
- 4393     3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 4394 of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 4395 mulation operator.

4396 Two domains are compatible with each other if values from one domain can be cast to values in  
 4397 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 4398 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 4399 any compatibility rule above is violated, execution of GrB\_extract ends and the domain mismatch  
 4400 error listed above is returned.

4401 From the arguments, the internal vector, matrix, mask, and index array used in the computation  
 4402 are formed ( $\leftarrow$  denotes copy):

- 4403     1. Vector  $\tilde{w} \leftarrow w$ .
- 4404     2. One-dimensional mask,  $\tilde{m}$ , is computed from argument **mask** as follows:  
 4405         (a) If **mask** = GrB\_NULL, then  $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$ .



4406 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,  
 4407 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,  
 4408 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .  
 4409 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .  
 4410 3. Matrix  $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .  
 4411 4. The internal row index array,  $\widetilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:  
 4412 (a) If `indices = GrB_ALL`, then  $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$ .  
 4413 (b) Otherwise,  $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nrows}$ .

4414 The internal vector, `mask`, and index array are checked for dimension compatibility. The following  
 4415 conditions must hold:

- 4416 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 4417 2.  $\text{size}(\widetilde{\mathbf{w}}) = \text{nrows}$ .

4418 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
 4419 match error listed above is returned.

4420 The `col_index` parameter is checked for a valid value. The following condition must hold:

- 4421 1.  $0 \leq \text{col\_index} < \text{ncols}(\mathbf{A})$

4422 If the rule above is violated, execution of `GrB_extract` ends and the invalid index error listed above  
 4423 is returned.

4424 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 4425 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4426 We are now ready to carry out the extract and any additional associated operations. We describe  
 4427 this in terms of two intermediate vectors:

- 4428 •  $\widetilde{\mathbf{t}}$ : The vector holding the extraction from a column of  $\widetilde{\mathbf{A}}$ .
- 4429 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4430 The intermediate vector,  $\widetilde{\mathbf{t}}$ , is created as follows:

$$4431 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}, \{(i, \widetilde{\mathbf{A}}(\widetilde{\mathbf{I}}[i], \text{col\_index})) \mid \forall i, 0 \leq i < \text{nrows} : (\widetilde{\mathbf{I}}[i], \text{col\_index}) \in \text{ind}(\widetilde{\mathbf{A}})\} \rangle.$$

4432 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`  
 4433 ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING` mode,  
 4434 the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the result  
 4435 vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4436 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

4437 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .

4438 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$4439 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4440 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
4441 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$4442 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

$$4443 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

$$4444 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

4445 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4446 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
4447 using what is called a *standard vector mask and replace*. This is carried out under control of the  
4448 mask which acts as a “write mask”.

4449 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
4450 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$4451 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4452 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
4453 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
4454 mask are unchanged:

$$4455 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4456 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
4457 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
4458 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but  
4459 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
4460 sequence.

### 4461 4.3.7 assign: Modifying sub-graphs

4462 Assign the contents of a subset of a matrix or vector.

#### 4463 4.3.7.1 assign: Standard vector variant

4464 Assign values from one GraphBLAS vector to a subset of a vector as specified by a set of indices.  
4465 The size of the input vector is the same size as the index array provided.

## 4468 C Syntax

```

4469         GrB_Info GrB_assign(GrB_Vector      w,
4470                             const GrB_Vector mask,
4471                             const GrB_BinaryOp accum,
4472                             const GrB_Vector u,
4473                             const GrB_Index *indices,
4474                             GrB_Index      nindices,
4475                             const GrB_Descriptor desc);

```

## 4476 Parameters

4477        **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
4478        that may be accumulated with the result of the assign operation. On output, this  
4479        vector holds the results of the operation.

4480        **mask** (IN) An optional “write” mask that controls which results from this operation are  
4481        stored into the output vector **w**. The mask dimensions must match those of the  
4482        vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4483        of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
4484        in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4485        dimensions of **w**), **GrB\_NULL** should be specified.

4486        **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
4487        entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4488        specified.

4489        **u** (IN) The GraphBLAS vector whose contents are assigned to a subset of **w**.

4490        **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
4491        **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0  
4492        to **nindices** – 1, then **GrB\_ALL** should be specified. Regardless of execution mode  
4493        and return value, this array may be manipulated by the caller after this operation  
4494        returns without affecting any deferred computations for this operation. If this  
4495        array contains duplicate values, it implies in assignment of more than one value to  
4496        the same location which leads to undefined results.

4497        **nindices** (IN) The number of values in **indices** array. Must be equal to **size(u)**.

4498        **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
4499        should be specified. Non-default field/value pairs are listed as follows:  
4500

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:

$$\begin{aligned}
 w(\text{indices}[i]) &= u(i), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\
 w(\text{indices}[i]) &= w(\text{indices}[i]) \odot u(i), \forall i : 0 \leq i < \text{nindices}.
 \end{aligned}$$

4529 Logically, this operation occurs in three steps:

4530     **Setup** The internal vectors and mask used in the computation are formed and their domains  
4531             and dimensions are tested for compatibility.

4532     **Compute** The indicated computations are carried out.

4533     **Output** The result is written into the output vector, possibly under control of a mask.

4534 Up to three argument vectors are used in the `GrB_assign` operation:

- 4535     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 4536     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4537     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4538 The argument vectors and the accumulation operator (if provided) are tested for domain compati-  
4539 bility as follows:

- 4540     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
4541         must be from one of the pre-defined types of Table 3.2.
- 4542     2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .
- 4543     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
4544         of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accu-  
4545         mulation operator.

4546 Two domains are compatible with each other if values from one domain can be cast to values in  
4547 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4548 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4549 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
4550 error listed above is returned.

4551 From the arguments, the internal vectors, mask and index array used in the computation are formed  
4552 ( $\leftarrow$  denotes copy):

- 4553     1. Vector  $\widetilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 4554     2. One-dimensional mask,  $\widetilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 4555         (a) If `mask` = `GrB_NULL`, then  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 4556         (b) If `mask`  $\neq$  `GrB_NULL`,
    - 4557             i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 4558             ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
  - 4559         (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .

4560 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

4561 4. The internal index array,  $\tilde{\mathbf{I}}$ , is computed from argument indices as follows:

4562 (a) If `indices = GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$ .

4563 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$ .

4564 The internal vector and mask are checked for dimension compatibility. The following conditions  
4565 must hold:

4566 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

4567 2.  $\text{nindices} = \text{size}(\tilde{\mathbf{u}})$ .

4568 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
4569 match error listed above is returned.

4570 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4571 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4572 We are now ready to carry out the assign and any additional associated operations. We describe  
4573 this in terms of two intermediate vectors:

- 4574 •  $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{w}}$ .
- 4575 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4576 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$4577 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nindices} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4578 At this point, if any value of  $\tilde{\mathbf{I}}[i]$  is outside the valid range of indices for vector  $\tilde{\mathbf{w}}$ , computation  
4579 ends and the method returns the index-out-of-bounds error listed above. In `GrB_NONBLOCKING`  
4580 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
4581 result vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4582 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

- 4583 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$4584 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4585 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
4586 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  
4587 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\text{ind}(\tilde{\mathbf{t}})$ ).

4588 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
4589 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$4590 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))),$$

$$4591 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4593 where the difference operator refers to set difference.

- If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$\langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ , using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a “write mask”.

- If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

- If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

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

#### 4.3.7.2 assign: Standard matrix variant

Assign values from one GraphBLAS matrix to a subset of a matrix as specified by a set of indices. The dimensions of the input matrix are the same size as the row and column index arrays provided.

## C Syntax

```
GrB_Info GrB_assign(GrB_Matrix      C,
                    const GrB_Matrix Mask,
                    const GrB_BinaryOp accum,
                    const GrB_Matrix A,
```

```

4627         const GrB_Index      *row_indices,
4628         GrB_Index             nrows,
4629         const GrB_Index      *col_indices,
4630         GrB_Index             ncols,
4631         const GrB_Descriptor desc);

```

## 4632 Parameters

4633     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
4634     that may be accumulated with the result of the assign operation. On output, the  
4635     matrix holds the results of the operation.

4636     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
4637     stored into the output matrix **C**. The mask dimensions must match those of the  
4638     matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4639     of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
4640     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4641     dimensions of **C**), **GrB\_NULL** should be specified.

4642     **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
4643     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4644     specified.

4645     **A** (IN) The GraphBLAS matrix whose contents are assigned to a subset of **C**.

4646     **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**  
4647     that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** – 1,  
4648     then **GrB\_ALL** can be specified. Regardless of execution mode and return value,  
4649     this array may be manipulated by the caller after this operation returns without  
4650     affecting any deferred computations for this operation. If this array contains du-  
4651     plicate values, it implies assignment of more than one value to the same location  
4652     which leads to undefined results.

4653     **nrows** (IN) The number of values in the **row\_indices** array. Must be equal to **nrows(A)**  
4654     if **A** is not transposed, or equal to **ncols(A)** if **A** is transposed.

4655     **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns  
4656     of **C** that are assigned. If all columns of **C** are to be assigned in order from 0  
4657     to **ncols** – 1, then **GrB\_ALL** should be specified. Regardless of execution mode  
4658     and return value, this array may be manipulated by the caller after this operation  
4659     returns without affecting any deferred computations for this operation. If this  
4660     array contains duplicate values, it implies assignment of more than one value to  
4661     the same location which leads to undefined results.

4662     **ncols** (IN) The number of values in **col\_indices** array. Must be equal to **ncols(A)** if **A** is  
4663     not transposed, or equal to **nrows(A)** if **A** is transposed.



4664  
4665  
4666

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:

4667

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.

## 4668 Return Values

4669  
4670  
4671  
4672  
4673

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.

4674

GrB\_PANIC Unknown internal error.

4675  
4676  
4677  
4678

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.

4679

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

4680  
4681

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).

4682  
4683  
4684

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.

4685  
4686

GrB\_DIMENSION\_MISMATCH Mask and C dimensions are incompatible, nrow  $\neq$  nrow(A), or ncols  $\neq$  ncols(A).

4687  
4688  
4689  
4690

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).

4691  
4692

GrB\_NULL\_POINTER Either argument row\_indices is a NULL pointer, argument col\_indices is a NULL pointer, or both.

## 4693 Description

4694 This variant of `GrB_assign` computes the result of assigning the contents of `A` to a subset of rows  
 4695 and columns in `C` in a specified order:  $C(\text{row\_indices}, \text{col\_indices}) = A$ ; or, if an optional binary  
 4696 accumulation operator ( $\odot$ ) is provided,  $C(\text{row\_indices}, \text{col\_indices}) = C(\text{row\_indices}, \text{col\_indices}) \odot$   
 4697 `A`. More explicitly (not accounting for an optional transpose of `A`):

$$\begin{aligned} C(\text{row\_indices}[i], \text{col\_indices}[j]) &= A(i, j), \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\ 4698 \quad C(\text{row\_indices}[i], \text{col\_indices}[j]) &= C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot A(i, j), \\ &\quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

4699 Logically, this operation occurs in three steps:

4700       Setup The internal matrices and mask used in the computation are formed and their domains  
 4701               and dimensions are tested for compatibility.

4702       Compute The indicated computations are carried out.

4703       Output The result is written into the output matrix, possibly under control of a mask.

4704 Up to three argument matrices are used in the `GrB_assign` operation:

- 4705       1.  $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4706       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)
- 4707       3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4708 The argument matrices and the accumulation operator (if provided) are tested for domain compat-  
 4709 ibility as follows:

- 4710       1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 4711               must be from one of the pre-defined types of Table 3.2.
- 4712       2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$ .
- 4713       3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 4714               of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 4715               mulation operator.

4716 Two domains are compatible with each other if values from one domain can be cast to values in  
 4717 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 4718 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 4719 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
 4720 error listed above is returned.

4721 From the arguments, the internal matrices, mask, and index arrays used in the computation are  
 4722 formed ( $\leftarrow$  denotes copy):

- 4723 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 4724 2. Two-dimensional mask  $\tilde{\mathbf{M}}$  is computed from argument `Mask` as follows:
- 4725 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
4726  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
- 4727 (b) If `Mask  $\neq$  GrB_NULL`,
- 4728 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
4729  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
- 4730 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
4731  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
- 4732 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 4733 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 4734 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
- 4735 (a) If `row_indices = GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .
- 4736 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{row\_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .
- 4737 5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:
- 4738 (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$ .
- 4739 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \mathbf{col\_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$ .

4740 The internal matrices and mask are checked for dimension compatibility. The following conditions  
4741 must hold:

- 4742 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 4743 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 4744 3.  $\mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}$ .
- 4745 4.  $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}$ .

4746 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
4747 match error listed above is returned.

4748 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4749 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4750 We are now ready to carry out the assign and any additional associated operations. We describe  
4751 this in terms of two intermediate vectors:

- 4752 •  $\tilde{\mathbf{T}}$ : The matrix holding the contents from  $\tilde{\mathbf{A}}$  in their destination locations relative to  $\tilde{\mathbf{C}}$ .
- 4753 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

4754 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$4755 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \tilde{\mathbf{A}}(i, j)) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols} : (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle.$$

4756 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  
 4757  $\tilde{\mathbf{J}}$  array is not in the range  $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$ , the execution of `GrB_assign` ends and the index out-of-  
 4758 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred  
 4759 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from  
 4760 this point forward in the sequence.

4761 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows:

- 4762 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}}$  is defined as

$$4763 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 4764 \quad \{(i, j, Z_{ij}) \mid \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4765 The above expression defines the structure of matrix  $\tilde{\mathbf{Z}}$  as follows: We start with the structure  
 4766 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  
 4767 assigned ( $\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$ ). Finally, we add the structure of  $\tilde{\mathbf{T}}$  ( $\mathbf{ind}(\tilde{\mathbf{T}})$ ).

4768 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 4769 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$4770 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4771 \\ 4772 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}),$$

4773 where the difference operator refers to set difference.

- 4774 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$4775 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4776 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 4777 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$4778 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 4779 \\ 4780 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4781 \\ 4782 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4783 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4784 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 4785 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 4786 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

4820 bool or any of the predefined “built-in” types in Table 3.2. If the default mask  
 4821 is desired (i.e., a mask that is all true with the dimensions of a column of C),  
 4822 GrB\_NULL should be specified.

4823 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
 4824 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
 4825 specified.

4826 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a column  
 4827 of C.

4828 **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
 4829 the specified column of C that are to be assigned. If all elements of the column  
 4830 in C are to be assigned in order from index 0 to **nrows** – 1, then GrB\_ALL should  
 4831 be specified. Regardless of execution mode and return value, this array may be  
 4832 manipulated by the caller after this operation returns without affecting any de-  
 4833 ferred computations for this operation. If this array contains duplicate values, it  
 4834 implies in assignment of more than one value to the same location which leads to  
 4835 undefined results.

4836 **nrows** (IN) The number of values in **row\_indices** array. Must be equal to **size(u)**.

4837 **col\_index** (IN) The index of the column in C to assign. Must be in the range [0, **ncols(C)**).

4838 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 4839 should be specified. Non-default field/value pairs are listed as follows:

4840

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output column in C is cleared (all elements removed) before result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.

4841

## 4842 Return Values

4843 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 4844 blocking mode, this indicates that the compatibility tests on  
 4845 dimensions and domains for the input arguments passed suc-  
 4846 cessfully. Either way, output matrix C is ready to be used in the  
 4847 next method of the sequence.

4848 **GrB\_PANIC** Unknown internal error.

4849           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
4850           opaque GraphBLAS objects (input or output) is in an invalid  
4851           state caused by a previous execution error. Call `GrB_error()` to  
4852           access any error messages generated by the implementation.

4853           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

4854           GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
4855           by a call to `new` (or `dup` for vector or matrix parameters).

4856           GrB\_INVALID\_INDEX `col_index` is outside the allowable range (i.e., greater than `ncols(C)`).

4857           GrB\_INDEX\_OUT\_OF\_BOUNDS A value in `row_indices` is greater than or equal to `nrows(C)`. In  
4858           non-blocking mode, this can be reported as an execution error.

4859           GrB\_DIMENSION\_MISMATCH `mask` size and number of rows in `C` are not the same, or `nrows`  $\neq$   
4860           `size(u)`.

4861           GrB\_DOMAIN\_MISMATCH The domains of the matrix and vector are incompatible with  
4862           each other or the corresponding domains of the accumulation  
4863           operator, or the mask's domain is not compatible with `bool` (in  
4864           the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

4865           GrB\_NULL\_POINTER Argument `row_indices` is a NULL pointer.

## 4866 Description

4867 This variant of `GrB_assign` computes the result of assigning a subset of locations in a column of a  
4868 GraphBLAS matrix (in a specific order) from the contents of a GraphBLAS vector:  
4869  $C(:, col\_index) = u$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C(:, col\_index) =$   
4870  $C(:, col\_index) \odot u$ . Taking order of `row_indices` into account, it is more explicitly written as:

$$4871 \quad C(row\_indices[i], col\_index) = u(i), \forall i : 0 \leq i < nrows, \text{ or}$$

$$C(row\_indices[i], col\_index) = C(row\_indices[i], col\_index) \odot u(i), \forall i : 0 \leq i < nrows.$$

4872 Logically, this operation occurs in three steps:

4873       **Setup** The internal matrices, vectors and mask used in the computation are formed and their  
4874       domains and dimensions are tested for compatibility.

4875       **Compute** The indicated computations are carried out.

4876       **Output** The result is written into the output matrix, possibly under control of a mask.

4877 Up to three argument vectors and matrices are used in this `GrB_assign` operation:

- 4878 1.  $C = \langle D(C), nrows(C), ncols(C), L(C) = \{(i, j, C_{ij})\} \rangle$
- 4879 2.  $mask = \langle D(mask), size(mask), L(mask) = \{(i, m_i)\} \rangle$  (optional)

4880 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4881 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain  
4882 compatibility as follows:

4883 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
4884 must be from one of the pre-defined types of Table 3.2.

4885 2.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .

4886 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
4887 of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
4888 mulation operator.

4889 Two domains are compatible with each other if values from one domain can be cast to values in  
4890 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4891 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4892 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
4893 error listed above is returned.

4894 The `col_index` parameter is checked for a valid value. The following condition must hold:

4895 1.  $0 \leq \text{col\_index} < \mathbf{ncols}(\mathbf{C})$

4896 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above  
4897 is returned.

4898 From the arguments, the internal vectors, `mask`, and index array used in the computation are  
4899 formed ( $\leftarrow$  denotes copy):

4900 1. The vector,  $\tilde{\mathbf{c}}$ , is extracted from a column of  $\mathbf{C}$  as follows:

4901 
$$\tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \{(i, C_{ij}) \mid i : 0 \leq i < \mathbf{nrows}(\mathbf{C}), j = \text{col\_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

4902 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:

4903 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{nrows}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C})\} \rangle$ .

4904 (b) If `mask`  $\neq$  `GrB_NULL`,

4905 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,

4906 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .

4907 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .

4908 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

4909 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:

4910 (a) If `row_indices` = `GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .



4911 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{row\_indices}[i]$ ,  $\forall i : 0 \leq i < \text{nrows}$ .

4912 The internal vectors, matrices, and masks are checked for dimension compatibility. The following  
4913 conditions must hold:

- 4914 1.  $\text{size}(\tilde{\mathbf{c}}) = \text{size}(\tilde{\mathbf{m}})$
- 4915 2.  $\text{nrows} = \text{size}(\tilde{\mathbf{u}})$ .

4916 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
4917 match error listed above is returned.

4918 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4919 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4920 We are now ready to carry out the assign and any additional associated operations. We describe  
4921 this in terms of two intermediate vectors:

- 4922 •  $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{c}}$ .
- 4923 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4924 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$4925 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nrows} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4926 At this point, if any value of  $\tilde{\mathbf{I}}[i]$  is outside the valid range of indices for vector  $\tilde{\mathbf{c}}$ , computation  
4927 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`  
4928 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
4929 result matrix,  $\mathbf{C}$ , is invalid from this point forward in the sequence.

4930 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

- 4931 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$4932 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4933 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
4934 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  
4935 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\text{ind}(\tilde{\mathbf{t}})$ ).

4936 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
4937 indices in  $\tilde{\mathbf{c}}$  and  $\tilde{\mathbf{t}}$ .

$$4938 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))),$$

$$4939 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4941 where the difference operator refers to set difference.

- If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$\langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$z_i = \tilde{\mathbf{c}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

Finally, the set of output values that make up the  $\tilde{\mathbf{z}}$  vector are written into the column of the final result matrix,  $\mathbf{C}(:, \text{col\_index})$ . This is carried out under control of the mask which acts as a “write mask”.

- If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}(:, \text{col\_index})$  on input to this operation are deleted and the new contents of the column is given by:

$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : j \neq \text{col\_index}\} \cup \{(i, \text{col\_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are copied into the column of the final result matrix,  $\mathbf{C}(:, \text{col\_index})$ , and elements of this column that fall outside the set indicated by the mask are unchanged:

$$\begin{aligned} \mathbf{L}(\mathbf{C}) = & \{(i, j, C_{ij}) : j \neq \text{col\_index}\} \cup \\ & \{(i, \text{col\_index}, \tilde{\mathbf{c}}(i)) : i \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \\ & \{(i, \text{col\_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}. \end{aligned}$$

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

#### 4.3.7.4 assign: Row variant

Assign the contents a vector to a subset of elements in one row of a matrix. Note that since the output cannot be transposed, a different variant of `assign` is provided to assign to a column of a matrix.

## 4972 C Syntax

```
4973         GrB_Info GrB_assign(GrB_Matrix      C,  
4974                             const GrB_Vector mask,  
4975                             const GrB_BinaryOp accum,  
4976                             const GrB_Vector u,  
4977                             GrB_Index      row_index,  
4978                             const GrB_Index *col_indices,  
4979                             GrB_Index      ncols,  
4980                             const GrB_Descriptor desc);
```

## 4981 Parameters

4982     **C** (INOUT) An existing GraphBLAS Matrix. On input, the matrix provides values  
4983     that may be accumulated with the result of the assign operation. On output, this  
4984     matrix holds the results of the operation.

4985     **mask** (IN) An optional “write” mask that controls which results from this operation are  
4986     stored into the specified row of the output matrix **C**. The mask dimensions must  
4987     match those of a single row of the matrix **C**. If the **GrB\_STRUCTURE** descriptor  
4988     is *not* set for the mask, the domain of the **Mask** matrix must be of type **bool** or  
4989     any of the predefined “built-in” types in Table 3.2. If the default mask is desired  
4990     (i.e., a mask that is all **true** with the dimensions of a row of **C**), **GrB\_NULL** should  
4991     be specified.

4992     **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
4993     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4994     specified.

4995     **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a row of  
4996     **C**.

4997     **row\_index** (IN) The index of the row in **C** to assign. Must be in the range  $[0, \mathbf{nrows}(\mathbf{C})]$ .

4998     **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
4999     the specified row of **C** that are to be assigned. If all elements of the row in **C** are to  
5000     be assigned in order from index 0 to  $\mathbf{ncols} - 1$ , then **GrB\_ALL** should be specified.  
5001     Regardless of execution mode and return value, this array may be manipulated by  
5002     the caller after this operation returns without affecting any deferred computations  
5003     for this operation. If this array contains duplicate values, it implies in assignment  
5004     of more than one value to the same location which leads to undefined results.

5005     **ncols** (IN) The number of values in **col\_indices** array. Must be equal to **size(u)**.

5006     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5007     should be specified. Non-default field/value pairs are listed as follows:  
5008

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:

5037  $C(\text{row\_index}, :) = u$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C(\text{row\_index}, :$   
 5038  $) = C(\text{row\_index}, :) \odot u$ . Taking order of `col_indices` into account it is more explicitly written as:

$$\begin{aligned} & C(\text{row\_index}, \text{col\_indices}[j]) = u(j), \forall j : 0 \leq j < \text{ncols}, \text{ or} \\ 5039 & C(\text{row\_index}, \text{col\_indices}[j]) = C(\text{row\_index}, \text{col\_indices}[j]) \odot u(j), \forall j : 0 \leq j < \text{ncols} \end{aligned}$$

5040 Logically, this operation occurs in three steps:

5041     **Setup** The internal matrices, vectors and mask used in the computation are formed and their  
 5042             domains and dimensions are tested for compatibility.

5043     **Compute** The indicated computations are carried out.

5044     **Output** The result is written into the output matrix, possibly under control of a mask.

5045 Up to three argument vectors and matrices are used in this `GrB_assign` operation:

- 5046     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5047     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 5048     3.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5049 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain  
 5050 compatibility as follows:

- 5051     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
 5052         must be from one of the pre-defined types of Table 3.2.
- 5053     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(u)$ .
- 5054     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 5055         of the accumulation operator and  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 5056         mulation operator.

5057 Two domains are compatible with each other if values from one domain can be cast to values in  
 5058 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 5059 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 5060 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
 5061 error listed above is returned.

5062 The `row_index` parameter is checked for a valid value. The following condition must hold:

- 5063     1.  $0 \leq \text{row\_index} < \mathbf{nrows}(C)$

5064 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above  
 5065 is returned.

5066 From the arguments, the internal vectors, mask, and index array used in the computation are  
 5067 formed ( $\leftarrow$  denotes copy):

5068 1. The vector,  $\tilde{\mathbf{c}}$ , is extracted from a row of  $\mathbf{C}$  as follows:

$$5069 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(j, C_{ij}) \mid \forall j : 0 \leq j < \mathbf{ncols}(\mathbf{C}), i = \text{row\_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

5070 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:

5071 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{ncols}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{ncols}(\mathbf{C})\} \rangle$ .

5072 (b) If `mask  $\neq$  GrB_NULL`,

5073 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,

5074 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .

5075 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .

5076 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

5077 4. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:

5078 (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$ .

5079 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \text{col\_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$ .

5080 The internal vectors, matrices, and masks are checked for dimension compatibility. The following  
5081 conditions must hold:

5082 1.  $\mathbf{size}(\tilde{\mathbf{c}}) = \mathbf{size}(\tilde{\mathbf{m}})$

5083 2.  $\mathbf{ncols} = \mathbf{size}(\tilde{\mathbf{u}})$ .

5084 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
5085 match error listed above is returned.

5086 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
5087 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5088 We are now ready to carry out the assign and any additional associated operations. We describe  
5089 this in terms of two intermediate vectors:

- 5090 •  $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{c}}$ .
- 5091 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

5092 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$5093 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{J}}[j], \tilde{\mathbf{u}}(j)) \mid \forall j, 0 \leq j < \mathbf{ncols} : j \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle.$$

5094 At this point, if any value of  $\tilde{\mathbf{J}}[j]$  is outside the valid range of indices for vector  $\tilde{\mathbf{c}}$ , computation  
5095 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`  
5096 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
5097 result matrix,  $\mathbf{C}$ , is invalid from this point forward in the sequence.

5098 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

5099 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$5100 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5101 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
5102 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  
5103 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\mathbf{ind}(\tilde{\mathbf{t}})$ ).

5104 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
5105 indices in  $\tilde{\mathbf{c}}$  and  $\tilde{\mathbf{t}}$ .

$$5106 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5107 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5109 where the difference operator refers to set difference.

5110 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$5111 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{c}}), \{(j, z_j) \mid j \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5112 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
5113 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$5114 \quad z_j = \tilde{\mathbf{c}}(j) \odot \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$5115 \quad z_j = \tilde{\mathbf{c}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5116 \quad z_j = \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

5118 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

5120 Finally, the set of output values that make up the  $\tilde{\mathbf{z}}$  vector are written into the column of the final  
5121 result matrix,  $\mathbf{C}(\text{row\_index}, :)$ . This is carried out under control of the mask which acts as a “write  
5122 mask”.

5123 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}(\text{row\_index}, :)$  on input to this  
5124 operation are deleted and the new contents of the column is given by:

$$5125 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row\_index}\} \cup \{(\text{row\_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5126 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
5127 copied into the column of the final result matrix,  $\mathbf{C}(\text{row\_index}, :)$ , and elements of this column  
5128 that fall outside the set indicated by the mask are unchanged:

$$5129 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row\_index}\} \cup$$

$$5130 \quad \{(\text{row\_index}, j, \tilde{\mathbf{c}}(j)) : j \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup$$

$$5131 \quad \{(\text{row\_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5132 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
5133 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
5134 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but may  
5135 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

5136 **4.3.7.5 assign: Constant vector variant**[Scott: NEW CONTENT]

5137 Assign the same value to a specified subset of vector elements. With the use of GrB\_ALL, the entire  
5138 destination vector can be filled with the constant.

5139 **C Syntax**

```
5140         GrB_Info GrB_assign(GrB_Vector          w,  
5141                             const GrB_Vector    mask,  
5142                             const GrB_BinaryOp   accum,  
5143                             <type>              val,  
5144                             const GrB_Index      *indices,  
5145                             GrB_Index            nindices,  
5146                             const GrB_Descriptor desc);
```

```
5147         GrB_Info GrB_assign(GrB_Vector          w,  
5148                             const GrB_Vector    mask,  
5149                             const GrB_BinaryOp   accum,  
5150                             const GrB_Scalar     s,  
5151                             const GrB_Index      *indices,  
5152                             GrB_Index            nindices,  
5153                             const GrB_Descriptor desc);
```

5154 **Parameters**

5155 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
5156 that may be accumulated with the result of the assign operation. On output, this  
5157 vector holds the results of the operation.

5158 **mask** (IN) An optional “write” mask that controls which results from this operation are  
5159 stored into the output vector w. The mask dimensions must match those of the  
5160 vector w. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
5161 of the mask vector must be of type bool or any of the predefined “built-in” types  
5162 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
5163 dimensions of w), GrB\_NULL should be specified.

5164 **accum** (IN) An optional binary operator used for accumulating entries into existing w  
5165 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
5166 specified.

5167 **val** (IN) Scalar value to assign to (a subset of) w.

5168 **s** (IN) Scalar value to assign to (a subset of) w.

5169 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
5170 w that are to be assigned. If all elements of w are to be assigned in order from 0



5171 to `nindices - 1`, then `GrB_ALL` should be specified. Regardless of execution mode  
 5172 and return value, this array may be manipulated by the caller after this operation  
 5173 returns without affecting any deferred computations for this operation. In this  
 5174 variant, the specific order of the values in the array has no effect on the result.  
 5175 Unlike other variants, if there are duplicated values in this array the result is still  
 5176 defined.

5177 **nindices** (IN) The number of values in `indices` array. Must be in the range: `[0, size(w)]`. If  
 5178 `nindices` is zero, the operation becomes a NO-OP.

5179 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
 5180 should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector <code>w</code> 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 <code>mask</code> vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of <code>mask</code> .

## 5183 Return Values

5184 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 5185 blocking mode, this indicates that the compatibility tests on  
 5186 dimensions and domains for the input arguments passed suc-  
 5187 cessfully. Either way, output vector `w` is ready to be used in the  
 5188 next method of the sequence.

5189 **GrB\_PANIC** Unknown internal error.

5190 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
 5191 opaque GraphBLAS objects (input or output) is in an invalid  
 5192 state caused by a previous execution error. Call `GrB_error()` to  
 5193 access any error messages generated by the implementation.

5194 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

5195 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized  
 5196 by a call to `new` (or `dup` for vector parameters).

5197 **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in `indices` is greater than or equal to `size(w)`. In non-  
 5198 blocking mode, this can be reported as an execution error.

5199 **GrB\_DIMENSION\_MISMATCH** `mask` and `w` dimensions are incompatible, or `nindices` is not less  
 5200 than `size(w)`.



5231 4. If **accum** is not **GrB\_NULL**, then either **D(val)** or **D(s)**, depending on the signature of the  
 5232 method, must be compatible with **D<sub>in2</sub>(accum)** of the accumulation operator.

5233 Two domains are compatible with each other if values from one domain can be cast to values in  
 5234 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 5235 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 5236 any compatibility rule above is violated, execution of **GrB\_assign** ends and the domain mismatch  
 5237 error listed above is returned.

5238 From the arguments, the internal vectors, mask and index array used in the computation are formed  
 5239 ( $\leftarrow$  denotes copy):

- 5240 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5241 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument **mask** as follows:
  - 5242 (a) If **mask** = **GrB\_NULL**, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 5243 (b) If **mask**  $\neq$  **GrB\_NULL**,
    - 5244 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 5245 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}(\mathbf{mask})(i) = \mathbf{true})\} \rangle$ .
  - 5246 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 5247 3. Scalar  $\tilde{s} \leftarrow s$  (**GrB\_Scalar** version only).
- 5248 4. The internal index array,  $\tilde{\mathbf{I}}$ , is computed from argument **indices** as follows:
  - 5249 (a) If **indices** = **GrB\_ALL**, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nindices}$ .
  - 5250 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{indices}[i], \forall i : 0 \leq i < \mathbf{nindices}$ .

5251 The internal vector and mask are checked for dimension compatibility. The following conditions  
 5252 must hold:

- 5253 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5254 2.  $0 \leq \mathbf{nindices} \leq \mathbf{size}(\tilde{\mathbf{w}})$ .

5255 If any compatibility rule above is violated, execution of **GrB\_assign** ends and the dimension mis-  
 5256 match error listed above is returned.

5257 From this point forward, in **GrB\_NONBLOCKING** mode, the method can optionally exit with  
 5258 **GrB\_SUCCESS** return code and defer any computation and/or execution error codes.

5259 We are now ready to carry out the assign and any additional associated operations. We describe  
 5260 this in terms of two intermediate vectors:

- 5261 •  $\tilde{\mathbf{t}}$ : The vector holding the copies of the scalar, either **val** or  $\tilde{s}$ , in their destination locations  
 5262 relative to  $\tilde{\mathbf{w}}$ .

5263 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

5264 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows. If a non-opaque scalar  $\mathbf{val}$  is provided:

$$5265 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{val}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5266 Correspondingly, if a non-empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 1$ ):

$$5267 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}(\tilde{s})) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5268 Finally, if an empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 0$ ):

$$5269 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \emptyset \rangle.$$

5270 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  
 5271 is not in the range  $[0, \mathbf{size}(\tilde{\mathbf{w}}))$ , the execution of `GrB_assign` ends and the index out-of-bounds  
 5272 error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a  
 5273 sequence-terminating `GrB_wait()` is called. Regardless, the result vector,  $\mathbf{w}$ , is invalid from this  
 5274 point forward in the sequence.

5275 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

5276 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$5277 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5278 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
 5279 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  
 5280 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\mathbf{ind}(\tilde{\mathbf{t}})$ ).

5281 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 5282 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$5283 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5284 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5286 where the difference operator refers to set difference. We note that in this case of assigning  
 5287 a constant,  $\{\tilde{\mathbf{I}}[k], \forall k\}$  and  $\mathbf{ind}(\tilde{\mathbf{t}})$  are identical.

5288 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$5289 \quad \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5290 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 5291 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$5292 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$5293 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5294 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

5297 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

5298 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 5299 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 5300 mask which acts as a “write mask”.

- 5301 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are  
 5302 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$5303 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 5304 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 5305 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 5306 mask are unchanged:

$$5307 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5308 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 5309 of vector  $\mathbf{w}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 5310 exits with return value GrB\_SUCCESS and the new content of vector  $\mathbf{w}$  is as defined above but  
 5311 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 5312 sequence.

#### 5313 4.3.7.6 assign: Constant matrix variant[Scott: NEW CONTENT]

5314 Assign the same value to a specified subset of matrix elements. With the use of GrB\_ALL, the  
 5315 entire destination matrix can be filled with the constant.

### 5316 C Syntax

```
5317      GrB_Info GrB_assign(GrB_Matrix      C,
5318                        const GrB_Matrix  Mask,
5319                        const GrB_BinaryOp accum,
5320                        <type>           val,
5321                        const GrB_Index    *row_indices,
5322                        GrB_Index          nrows,
5323                        const GrB_Index    *col_indices,
5324                        GrB_Index          ncols,
5325                        const GrB_Descriptor desc);
```

```
5326      GrB_Info GrB_assign(GrB_Matrix      C,
5327                        const GrB_Matrix  Mask,
5328                        const GrB_BinaryOp accum,
5329                        const GrB_Scalar   s,
5330                        const GrB_Index    *row_indices,
5331                        GrB_Index          nrows,
```

```

5332         const GrB_Index      *col_indices,
5333         GrB_Index             ncols,
5334         const GrB_Descriptor  desc);

```

## 5335 Parameters

5336       **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
5337       that may be accumulated with the result of the assign operation. On output, the  
5338       matrix holds the results of the operation.

5339       **Mask** (IN) An optional “write” mask that controls which results from this operation are  
5340       stored into the output matrix **C**. The mask dimensions must match those of the  
5341       matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5342       of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
5343       in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5344       dimensions of **C**), **GrB\_NULL** should be specified.

5345       **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
5346       entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5347       specified.

5348       **val** (IN) Scalar value to assign to (a subset of) **C**.

5349       **s** (IN) Scalar value to assign to (a subset of) **C**.

5350       **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**  
5351       that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** – 1,  
5352       then **GrB\_ALL** can be specified. Regardless of execution mode and return value,  
5353       this array may be manipulated by the caller after this operation returns without  
5354       affecting any deferred computations for this operation. Unlike other variants, if  
5355       there are duplicated values in this array the result is still defined.

5356       **nrows** (IN) The number of values in **row\_indices** array. Must be in the range: [0, **nrows**(**C**)].  
5357       If **nrows** is zero, the operation becomes a NO-OP.

5358       **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns of **C**  
5359       that are assigned. If all columns of **C** are to be assigned in order from 0 to **ncols** – 1,  
5360       then **GrB\_ALL** should be specified. Regardless of execution mode and return value,  
5361       this array may be manipulated by the caller after this operation returns without  
5362       affecting any deferred computations for this operation. Unlike other variants, if  
5363       there are duplicated values in this array the result is still defined.

5364       **ncols** (IN) The number of values in **col\_indices** array. Must be in the range: [0, **ncols**(**C**)].  
5365       If **ncols** is zero, the operation becomes a NO-OP.

5366       **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5367       should be specified. Non-default field/value pairs are listed as follows:  
5368

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`

5398 or  $C(\text{row\_indices}, \text{col\_indices}) = s$  is performed. If an optional binary accumulation operator  
 5399  $(\odot)$  is provided, then either  $C(\text{row\_indices}, \text{col\_indices}) = C(\text{row\_indices}, \text{col\_indices}) \odot \text{val}$  or  
 5400  $C(\text{row\_indices}, \text{col\_indices}) = C(\text{row\_indices}, \text{col\_indices}) \odot s$  is performed. More explicitly, if a  
 5401 non-opaque value  $\text{val}$  is provided:

$$\begin{aligned} & C(\text{row\_indices}[i], \text{col\_indices}[j]) = \text{val}, \text{ or} \\ 5402 \quad & C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot \text{val} \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5403 Correspondingly, if a `GrB_Scalar`  $s$  is provided:

$$\begin{aligned} & C(\text{row\_indices}[i], \text{col\_indices}[j]) = s, \text{ or} \\ 5404 \quad & C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot s \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5405 Logically, this operation occurs in three steps:

5406     Setup The internal vectors and mask used in the computation are formed and their domains  
 5407             and dimensions are tested for compatibility.

5408     Compute The indicated computations are carried out.

5409     Output The result is written into the output matrix, possibly under control of a mask.

5410 Up to two argument matrices are used in the `GrB_assign` operation:

- 5411     1.  $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5412     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)

5413 The argument scalar, matrices, and the accumulation operator (if provided) are tested for domain  
 5414 compatibility as follows:

- 5415     1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 5416         must be from one of the pre-defined types of Table 3.2.
- 5417     2.  $\mathbf{D}(C)$  must be compatible with either  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(s)$ , depending on the signature of the  
 5418         method.
- 5419     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 5420         of the accumulation operator.
- 5421     4. If `accum` is not `GrB_NULL`, then either  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(s)$ , depending on the signature of the  
 5422         method, must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.



5423 Two domains are compatible with each other if values from one domain can be cast to values in  
 5424 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 5425 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 5426 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
 5427 error listed above is returned.

5428 From the arguments, the internal matrices, index arrays, and mask used in the computation are  
 5429 formed ( $\leftarrow$  denotes copy):

- 5430 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 5431 2. Two-dimensional mask  $\tilde{\mathbf{M}}$  is computed from argument `Mask` as follows:
  - 5432 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 5433  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 5434 (b) If `Mask  $\neq$  GrB_NULL`,
    - 5435 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
 5436  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 5437 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
 5438  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 5439 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 5440 3. Scalar  $\tilde{s} \leftarrow s$  (`GrB_Scalar` version only).
- 5441 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
  - 5442 (a) If `row_indices = GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .
  - 5443 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{row\_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .
- 5444 5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:
  - 5445 (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$ .
  - 5446 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \mathbf{col\_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$ .

5447 The internal matrix and mask are checked for dimension compatibility. The following conditions  
 5448 must hold:

- 5449 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 5450 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 5451 3.  $0 \leq \mathbf{nrows} \leq \mathbf{nrows}(\tilde{\mathbf{C}})$ .
- 5452 4.  $0 \leq \mathbf{ncols} \leq \mathbf{ncols}(\tilde{\mathbf{C}})$ .

5453 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
 5454 match error listed above is returned.

5455 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 5456 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5457 We are now ready to carry out the assign and any additional associated operations. We describe  
 5458 this in terms of two intermediate matrices:

- 5459 •  $\tilde{\mathbf{T}}$ : The matrix holding the copies of the scalar, either `val` or  $\tilde{s}$ , in their destination locations  
 5460 relative to  $\tilde{\mathbf{C}}$ .
- 5461 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

5462 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows. If a non-opaque scalar `val` is provided:

$$5463 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\text{val}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle.$$

5464 Correspondingly, if a non-empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e., `size`( $\tilde{s}$ ) = 1):

$$5465 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}(\tilde{s})) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle.$$

5466 Finally, if an empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e., `size`( $\tilde{s}$ ) = 0):

$$5467 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \emptyset \rangle.$$

5468 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  
 5469 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  
 5470  $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$ , the execution of `GrB_assign` ends and the index out-of-bounds error listed above is  
 5471 generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating  
 5472 `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from this point forward in the  
 5473 sequence.

5474 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows:

- 5475 • If `accum` = `GrB_NULL`, then  $\tilde{\mathbf{Z}}$  is defined as

$$5476 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 5477 \quad \{(i, j, Z_{ij}) \mid (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}}))\} \rangle.$$

5478 The above expression defines the structure of matrix  $\tilde{\mathbf{Z}}$  as follows: We start with the structure  
 5479 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  
 5480 assigned ( $\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$ ). Finally, we add the structure of  $\tilde{\mathbf{T}}$  ( $\mathbf{ind}(\tilde{\mathbf{T}})$ ).

5481 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 5482 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$5483 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5484 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}),$$

5486 where the difference operator refers to set difference. We note that, in this particular case of  
 5487 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.

- If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$\langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

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.

### 4.3.8 apply: Apply a function to the elements of an object

Computes the transformation of the values of the elements of a vector or a matrix using a unary function, or a binary function where one argument is bound to a scalar.

#### 4.3.8.1 apply: Vector variant

Computes the transformation of the values of the elements of a vector using a unary function.

## 5518 C Syntax

```

5519         GrB_Info GrB_apply(GrB_Vector          w,
5520                             const GrB_Vector    mask,
5521                             const GrB_BinaryOp   accum,
5522                             const GrB_UnaryOp    op,
5523                             const GrB_Vector    u,
5524                             const GrB_Descriptor desc);

```

## 5525 Parameters

5526     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
5527     that may be accumulated with the result of the apply operation. On output, this  
5528     vector holds the results of the operation.

5529     **mask** (IN) An optional “write” mask that controls which results from this operation are  
5530     stored into the output vector **w**. The mask dimensions must match those of the  
5531     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5532     of the mask vector must be of type **bool** or any of the predefined “built-in” types  
5533     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5534     dimensions of **w**), **GrB\_NULL** should be specified.

5535     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
5536     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5537     specified.

5538     **op** (IN) A unary operator applied to each element of input vector **u**.

5539     **u** (IN) The GraphBLAS vector to which the unary function is applied.

5540     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5541     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> .

## 5544 Return Values

5545     **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
5546     blocking mode, this indicates that the compatibility tests on di-  
5547     mensions and domains for the input arguments passed successfully.

5548                                Either way, output vector  $w$  is ready to be used in the next method  
5549                                of the sequence.

5550                                **GrB\_PANIC** Unknown internal error.

5551                                **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
5552                                GraphBLAS objects (input or output) is in an invalid state caused  
5553                                by a previous execution error. Call **GrB\_error()** to access any error  
5554                                messages generated by the implementation.

5555                                **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

5556 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
5557                                a call to **new** (or **dup** for vector parameters).

5558 **GrB\_DIMENSION\_MISMATCH**  $mask$ ,  $w$  and/or  $u$  dimensions are incompatible.

5559                                **GrB\_DOMAIN\_MISMATCH** The domains of the various vectors are incompatible with the corre-  
5560                                sponding domains of the accumulation operator or unary function,  
5561                                or the mask's domain is not compatible with **bool** (in the case where  
5562                                 $desc[GrB\_MASK].GrB\_STRUCTURE$  is not set).

## 5563 **Description**

5564    This variant of **GrB\_apply** computes the result of applying a unary function to the elements of a  
5565    GraphBLAS vector:  $w = f(u)$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  
5566     $w = w \odot f(u)$ .

5567    Logically, this operation occurs in three steps:

5568                                **Setup** The internal vectors and mask used in the computation are formed and their domains  
5569                                and dimensions are tested for compatibility.

5570                                **Compute** The indicated computations are carried out.

5571                                **Output** The result is written into the output vector, possibly under control of a mask.

5572    Up to three argument vectors are used in this **GrB\_apply** operation:

- 5573        1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 5574        2.  $mask = \langle \mathbf{D}(mask), \mathbf{size}(mask), \mathbf{L}(mask) = \{(i, m_i)\} \rangle$  (optional)
- 5575        3.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5576    The argument vectors, unary operator and the accumulation operator (if provided) are tested for  
5577    domain compatibility as follows:

- 5578 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
5579 must be from one of the pre-defined types of Table 3.2.
- 5580 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the unary operator.
- 5581 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
5582 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the unary operator must be compatible with  
5583  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 5584 4.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in}(\text{op})$ .

5585 Two domains are compatible with each other if values from one domain can be cast to values in  
5586 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
5587 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
5588 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
5589 error listed above is returned.

5590 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
5591 denotes copy):

- 5592 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5593 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 5594 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .
  - 5595 (b) If `mask`  $\neq$  `GrB_NULL`,
    - 5596 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
    - 5597 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 5598 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 5599 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

5600 The internal vectors and masks are checked for dimension compatibility. The following conditions  
5601 must hold:

- 5602 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$
- 5603 2.  $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$ .

5604 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
5605 error listed above is returned.

5606 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
5607 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5608 We are now ready to carry out the apply and any additional associated operations. We describe  
5609 this in terms of two intermediate vectors:

- 5610 •  $\tilde{\mathbf{t}}$ : The vector holding the result from applying the unary operator to the input vector  $\tilde{\mathbf{u}}$ .
- 5611 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

5612 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$5613 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f(\tilde{\mathbf{u}}(i))) \mid \forall i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle,$$

5614 where  $f = \mathbf{f}(\text{op})$ .

5615 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 5616 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 5617 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$5618 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5619 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
5620 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 5621 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 5622 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 5623 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 5624 \end{aligned}$$

5625 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

5627 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
5628 using what is called a *standard vector mask and replace*. This is carried out under control of the  
5629 mask which acts as a “write mask”.

- 5630 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
5631 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$5632 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

- 5633 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
5634 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
5635 mask are unchanged:

$$5636 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

5637 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
5638 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
5639 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but  
5640 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
5641 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.



## 5670 Return Values

5671	<b>GrB_SUCCESS</b>	In blocking mode, the operation completed successfully. In non-
5672		blocking mode, this indicates that the compatibility tests on
5673		dimensions and domains for the input arguments passed suc-
5674		cessfully. Either way, output matrix C is ready to be used in the
5675		next method of the sequence.
5676	<b>GrB_PANIC</b>	Unknown internal error.
5677	<b>GrB_INVALID_OBJECT</b>	This is returned in any execution mode whenever one of the
5678		opaque GraphBLAS objects (input or output) is in an invalid
5679		state caused by a previous execution error. Call <b>GrB_error()</b> to
5680		access any error messages generated by the implementation.
5681	<b>GrB_OUT_OF_MEMORY</b>	Not enough memory available for the operation.
5682	<b>GrB_UNINITIALIZED_OBJECT</b>	One or more of the GraphBLAS objects has not been initialized
5683		by a call to <b>new</b> (or <b>Matrix_dup</b> for matrix parameters).
5684	<b>GrB_DIMENSION_MISMATCH</b>	Mask and C dimensions are incompatible, <b>nrows</b> $\neq$ <b>nrows</b> (C), or
5685		<b>ncols</b> $\neq$ <b>ncols</b> (C).
5686	<b>GrB_DOMAIN_MISMATCH</b>	The domains of the various matrices are incompatible with the
5687		corresponding domains of the accumulation operator or unary
5688		function, or the mask's domain is not compatible with <b>bool</b> (in
5689		the case where <b>desc</b> [ <b>GrB_MASK</b> ]. <b>GrB_STRUCTURE</b> is not set).

## 5690 Description

5691 This variant of **GrB\_apply** computes the result of applying a unary function to the elements of a  
 5692 GraphBLAS matrix:  $C = f(A)$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  
 5693  $C = C \odot f(A)$ .

5694 Logically, this operation occurs in three steps:

5695     **Setup** The internal matrices and mask used in the computation are formed and their domains  
 5696     and dimensions are tested for compatibility.

5697     **Compute** The indicated computations are carried out.

5698     **Output** The result is written into the output matrix, possibly under control of a mask.

5699 Up to three argument matrices are used in the **GrB\_apply** operation:

- 5700 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5701 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)

5702 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

5703 The argument matrices, unary operator and the accumulation operator (if provided) are tested for  
5704 domain compatibility as follows:

- 5705 1. If **Mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\text{Mask})$   
5706 must be from one of the pre-defined types of Table 3.2.
- 5707 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the unary operator.
- 5708 3. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
5709 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the unary operator must be compatible with  
5710  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 5711 4.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in}(\text{op})$  of the unary operator.

5712 Two domains are compatible with each other if values from one domain can be cast to values in  
5713 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
5714 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
5715 any compatibility rule above is violated, execution of **GrB\_apply** ends and the domain mismatch  
5716 error listed above is returned.

5717 From the argument matrices, the internal matrices, mask, and index arrays used in the computation  
5718 are formed ( $\leftarrow$  denotes copy):

- 5719 1. Matrix  $\tilde{C} \leftarrow C$ .
- 5720 2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument **Mask** as follows:
  - 5721 (a) If **Mask** = **GrB\_NULL**, then  $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$   
5722  $j < \mathbf{ncols}(C)\} \rangle$ .
  - 5723 (b) If **Mask**  $\neq$  **GrB\_NULL**,
    - 5724 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
5725  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
    - 5726 ii. Otherwise,  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
5727  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
  - 5728 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\tilde{M} \leftarrow \neg \tilde{M}$ .
- 5729 3. Matrix  $\tilde{A} \leftarrow \text{desc[GrB_INP0].GrB_TRAN} ? A^T : A$ .

5730 The internal matrices and mask are checked for dimension compatibility. The following conditions  
5731 must hold:

- 5732 1.  $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{M})$ .
- 5733 2.  $\mathbf{ncols}(\tilde{C}) = \mathbf{ncols}(\tilde{M})$ .
- 5734 3.  $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{A})$ .

5735 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

5736 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
5737 error listed above is returned.

5738 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
5739 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5740 We are now ready to carry out the apply and any additional associated operations. We describe  
5741 this in terms of two intermediate matrices:

- 5742 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the unary operator to the input matrix  $\tilde{\mathbf{A}}$ .
- 5743 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

5744 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$5745 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j))) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

5746 where  $f = \mathbf{f}(\mathbf{op})$ .

5747 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 5748 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 5749 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$5750 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5751 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
5752 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$5753 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$5754 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$5755 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5756 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

5759 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
5760 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
5761 mask which acts as a “write mask”.

- 5762 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
5763 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$5764 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\text{ind}(\mathbf{C}) \cap \text{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\text{ind}(\tilde{\mathbf{Z}}) \cap \text{ind}(\tilde{\mathbf{M}}))\}.$$

In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

#### 4.3.8.3 apply: Vector-BinaryOp variants[Scott: NEW CONTENT]

Computes the transformation of the values of the stored elements of a vector using a binary operator and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the first argument to the binary operator and stored elements of the vector are passed as the second argument. In the *bind-second* variant, the elements of the vector are passed as the first argument and the specified scalar value is passed as the second argument. The scalar can be passed either as a non-opaque variable or as a GrB\_Scalar object.

#### C Syntax

```
// bind-first + scalar value
GrB_Info GrB_apply(GrB_Vector          w,
                  const GrB_Vector      mask,
                  const GrB_BinaryOp    accum,
                  const GrB_BinaryOp    op,
                  <type>                val,
                  const GrB_Vector      u,
                  const GrB_Descriptor  desc);
```

```
// bind-first + GraphBLAS scalar
GrB_Info GrB_apply(GrB_Vector          w,
                  const GrB_Vector      mask,
                  const GrB_BinaryOp    accum,
                  const GrB_BinaryOp    op,
                  const GrB_Scalar      s,
                  const GrB_Vector      u,
                  const GrB_Descriptor  desc);
```

```
// bind-second + scalar value
GrB_Info GrB_apply(GrB_Vector          w,
                  const GrB_Vector      mask,
```

```

5801         const GrB_BinaryOp      accum,
5802         const GrB_BinaryOp      op,
5803         const GrB_Vector        u,
5804         <type>                  val,
5805         const GrB_Descriptor    desc);

5806 // bind-second + GraphBLAS scalar
5807 GrB_Info GrB_apply(GrB_Vector      w,
5808                   const GrB_Vector mask,
5809                   const GrB_BinaryOp accum,
5810                   const GrB_BinaryOp op,
5811                   const GrB_Vector u,
5812                   const GrB_Scalar s,
5813                   const GrB_Descriptor desc);

```

## 5814 Parameters

5815     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
5816     that may be accumulated with the result of the apply operation. On output, this  
5817     vector holds the results of the operation.

5818     **mask** (IN) An optional “write” mask that controls which results from this operation are  
5819     stored into the output vector **w**. The mask dimensions must match those of the  
5820     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5821     of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
5822     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5823     dimensions of **w**), **GrB\_NULL** should be specified.

5824     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
5825     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5826     specified.

5827     **op** (IN) A binary operator applied to each element of input vector, **u**, and the scalar  
5828     value, **val**.

5829     **u** (IN) The GraphBLAS vector whose elements are passed to the binary operator as  
5830     the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)  
5831     argument in the *bind-second* variant.

5832     **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)  
5833     argument in the *bind-first* variant, or the right-hand (second) argument in the  
5834     *bind-second* variant.

5835     **s** (IN) A GraphBLAS scalar that is passed to the binary operator as the left-hand  
5836     (first) argument in the *bind-first* variant, or the right-hand (second) argument in  
5837     the *bind-second* variant. It must not be empty.

5838 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
5839 should be specified. Non-default field/value pairs are listed as follows:

5840

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> .

5841

## 5842 Return Values

5843 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
5844 blocking mode, this indicates that the compatibility tests on di-  
5845 mensions and domains for the input arguments passed successfully.  
5846 Either way, output vector **w** is ready to be used in the next method  
5847 of the sequence.

5848 GrB\_PANIC Unknown internal error.

5849 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
5850 GraphBLAS objects (input or output) is in an invalid state caused  
5851 by a previous execution error. Call GrB\_error() to access any error  
5852 messages generated by the implementation.

5853 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

5854 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
5855 a call to new (or dup for vector parameters).

5856 GrB\_DIMENSION\_MISMATCH mask, w and/or u dimensions are incompatible.

5857 GrB\_DOMAIN\_MISMATCH The domains of the various vectors and scalar are incompatible with  
5858 the corresponding domains of the binary operator or accumulation  
5859 operator, or the mask's domain is not compatible with bool (in the  
5860 case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

5861 GrB\_EMPTY\_OBJECT The GrB\_Scalar **s** used in the call is empty (**nvals(s) = 0**) and  
5862 therefore a value cannot be passed to the binary operator.

## 5863 Description

5864 This variant of GrB\_apply computes the result of applying a binary operator to the elements of a  
5865 GraphBLAS vector each composed with a scalar constant, either **val** or **s**:

5866                   bind-first:      $w = f(\text{val}, u)$  or  $w = f(s, u)$

5867                   bind-second:     $w = f(u, \text{val})$  or  $w = f(u, s)$ ,

5868 or if an optional binary accumulation operator ( $\odot$ ) is provided:

5869                   bind-first:      $w = w \odot f(\text{val}, u)$  or  $w = w \odot f(s, u)$

5870                   bind-second:     $w = w \odot f(u, \text{val})$  or  $w = w \odot f(u, s)$ .

5871 Logically, this operation occurs in three steps:

5872     **Setup** The internal vectors and mask used in the computation are formed and their domains  
5873             and dimensions are tested for compatibility.

5874     **Compute** The indicated computations are carried out.

5875     **Output** The result is written into the output vector, possibly under control of a mask.

5876 Up to three argument vectors are used in this GrB\_apply operation:

5877     1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$

5878     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)

5879     3.  $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5880 The argument scalar, vectors, binary operator and the accumulation operator (if provided) are  
5881 tested for domain compatibility as follows:

5882     1. If **mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{mask})$   
5883         must be from one of the pre-defined types of Table 3.2.

5884     2.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the binary operator.

5885     3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
5886         of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the binary operator must be compatible with  
5887          $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.

5888     4.  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.

5889     5. If bind-first:

5890         (a)  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the binary operator.

5891         (b) If the non-opaque scalar **val** is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$   
5892             of the binary operator.

5893         (c) If the GrB\_Scalar **s** is provided, then  $\mathbf{D}(s)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the  
5894             binary operator.

- 5895 6. If bind-second:
- 5896 (a)  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the binary operator.
- 5897 (b) If the non-opaque scalar  $\mathbf{val}$  is provided, then  $\mathbf{D}(\mathbf{val})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$
- 5898 of the binary operator.
- 5899 (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
- 5900 binary operator.

5901 Two domains are compatible with each other if values from one domain can be cast to values in

5902 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all

5903 compatible with each other. A domain from a user-defined type is only compatible with itself. If

5904 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch

5905 error listed above is returned.

5906 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$

5907 denotes copy):

- 5908 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5909 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
- 5910 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
- 5911 (b) If `mask  $\neq$  GrB_NULL`,
- 5912 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
- 5913 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
- 5914 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 5915 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 5916 4. Scalar  $\tilde{\mathbf{s}} \leftarrow \mathbf{s}$  (GraphBLAS scalar case).

5917 The internal vectors and masks are checked for dimension compatibility. The following conditions

5918 must hold:

- 5919 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5920 2.  $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{w}})$ .

5921 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch

5922 error listed above is returned.

5923 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with

5924 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5925 If an empty `GrB_Scalar`  $\tilde{\mathbf{s}}$  is provided ( $\mathbf{nvals}(\tilde{\mathbf{s}}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.

5926 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

5927 `val` with the same domain as  $\tilde{\mathbf{s}}$  and set `val = val( $\tilde{\mathbf{s}}$ )`.

5928 We are now ready to carry out the apply and any additional associated operations. We describe

5929 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.

#### 5963 4.3.8.4 apply: Matrix-BinaryOp variants[Scott: NEW CONTENT]

5964 Computes the transformation of the values of the stored elements of a matrix using a binary  
5965 operator and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the  
5966 first argument to the binary operator and stored elements of the matrix are passed as the second  
5967 argument. In the *bind-second* variant, the elements of the matrix are passed as the first argument  
5968 and the specified scalar value is passed as the second argument. The scalar can be passed either as  
5969 a non-opaque variable or as a GrB\_Scalar object.

#### 5970 C Syntax

```
5971 // bind-first + scalar value
5972 GrB_Info GrB_apply(GrB_Matrix      C,
5973                   const GrB_Matrix Mask,
5974                   const GrB_BinaryOp accum,
5975                   const GrB_BinaryOp op,
5976                   <type>           val,
5977                   const GrB_Matrix A,
5978                   const GrB_Descriptor desc);
```

```
5979 // bind-first + GraphBLAS scalar
5980 GrB_Info GrB_apply(GrB_Matrix      C,
5981                   const GrB_Matrix Mask,
5982                   const GrB_BinaryOp accum,
5983                   const GrB_BinaryOp op,
5984                   const GrB_Scalar s,
5985                   const GrB_Matrix A,
5986                   const GrB_Descriptor desc);
```

```
5987 // bind-second + scalar value
5988 GrB_Info GrB_apply(GrB_Matrix      C,
5989                   const GrB_Matrix Mask,
5990                   const GrB_BinaryOp accum,
5991                   const GrB_BinaryOp op,
5992                   const GrB_Matrix A,
5993                   <type>           val,
5994                   const GrB_Descriptor desc);
```

```
5995 // bind-second + GraphBLAS scalar
5996 GrB_Info GrB_apply(GrB_Matrix      C,
5997                   const GrB_Matrix Mask,
5998                   const GrB_BinaryOp accum,
5999                   const GrB_BinaryOp op,
6000                   const GrB_Matrix A,
```

```

6001         const GrB_Scalar      s,
6002         const GrB_Descriptor desc);

```

## 6003 Parameters

6004     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
6005     that may be accumulated with the result of the apply operation. On output, the  
6006     matrix holds the results of the operation.

6007     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
6008     stored into the output matrix C. The mask dimensions must match those of the  
6009     matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
6010     of the Mask matrix must be of type `bool` or any of the predefined “built-in” types  
6011     in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
6012     dimensions of C), `GrB_NULL` should be specified.

6013     **accum** (IN) An optional binary operator used for accumulating entries into existing C  
6014     entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
6015     specified.

6016     **op** (IN) A binary operator applied to each element of input matrix, A, with the element  
6017     of the input matrix used as the left-hand argument, and the scalar value, `val`, used  
6018     as the right-hand argument.

6019     **A** (IN) The GraphBLAS matrix whose elements are passed to the binary operator as  
6020     the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)  
6021     argument in the *bind-second* variant.

6022     **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)  
6023     argument in the *bind-first* variant, or the right-hand (second) argument in the  
6024     *bind-second* variant.

6025     **s** (IN) GraphBLAS scalar value that is passed to the binary operator as the left-hand  
6026     (first) argument in the *bind-first* variant, or the right-hand (second) argument in  
6027     the *bind-second* variant. It must not be empty.

6028     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
6029     should be specified. Non-default field/value pairs are listed as follows:  
6030

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.

## 6058 Description

6059 This variant of `GrB_apply` computes the result of applying a binary operator to the elements of a  
6060 GraphBLAS matrix each composed with a scalar constant, `val` or `s`:

6061                   bind-first:      $C = f(\text{val}, A)$  or  $C = f(s, A)$

6062                   bind-second:     $C = f(A, \text{val})$  or  $C = f(A, s)$ ,

6063 or if an optional binary accumulation operator ( $\odot$ ) is provided:

6064                   bind-first:      $C = C \odot f(\text{val}, A)$  or  $C = C \odot f(s, A)$

6065                   bind-second:     $C = C \odot f(A, \text{val})$  or  $C = C \odot f(A, s)$ .

6066 Logically, this operation occurs in three steps:

6067       **Setup** The internal matrices and mask used in the computation are formed and their domains  
6068               and dimensions are tested for compatibility.

6069       **Compute** The indicated computations are carried out.

6070       **Output** The result is written into the output matrix, possibly under control of a mask.

6071 Up to three argument matrices are used in the `GrB_apply` operation:

6072     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$

6073     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)

6074     3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6075 The argument scalar, matrices, binary operator and the accumulation operator (if provided) are  
6076 tested for domain compatibility as follows:

6077     1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
6078       must be from one of the pre-defined types of Table 3.2.

6079     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the binary operator.

6080     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
6081       of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the binary operator must be compatible with  
6082        $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.

6083     4.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.

6084     5. If bind-first:

6085       (a)  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the binary operator.

6086 (b) If the non-opaque scalar  $\text{val}$  is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$   
 6087 of the binary operator.

6088 (c) If the `GrB_Scalar`  $s$  is provided, then  $\mathbf{D}(s)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the  
 6089 binary operator.

6090 6. If `bind-second`:

6091 (a)  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.

6092 (b) If the non-opaque scalar  $\text{val}$  is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$   
 6093 of the binary operator.

6094 (c) If the `GrB_Scalar`  $s$  is provided, then  $\mathbf{D}(s)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the  
 6095 binary operator.

6096 Two domains are compatible with each other if values from one domain can be cast to values in  
 6097 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 6098 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 6099 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
 6100 error listed above is returned.

6101 From the argument matrices, the internal matrices, mask, and index arrays used in the computation  
 6102 are formed ( $\leftarrow$  denotes copy):

6103 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .

6104 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:

6105 (a) If `Mask` = `GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 6106  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .

6107 (b) If `Mask`  $\neq$  `GrB_NULL`,

6108 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
 6109  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,

6110 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
 6111  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .

6112 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .

6113 3. Matrix  $\tilde{\mathbf{A}}$  is computed from argument `A` as follows:

6114 `bind-first`:  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP1}].\text{GrB\_TRAN} ? A^T : A$

6115 `bind-second`:  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? A^T : A$

6116 4. Scalar  $\tilde{s} \leftarrow s$  (`GraphBLAS` scalar case).

6117 The internal matrices and mask are checked for dimension compatibility. The following conditions  
 6118 must hold:

6119 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .

6120 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .

6121 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .

6122 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

6123 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6124 error listed above is returned.

6125 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6126 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6127 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6128 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable  
6129  $\mathbf{val}$  with the same domain as  $\tilde{s}$  and set  $\mathbf{val} = \mathbf{val}(\tilde{s})$ .

6130 We are now ready to carry out the apply and any additional associated operations. We describe  
6131 this in terms of two intermediate matrices:

- 6132 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the binary operator to the input matrix  $\tilde{\mathbf{A}}$ .
- 6133 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6134 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as one of the following:

6135 bind-first:  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\mathbf{val}, \tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

6136 bind-second:  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j), \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

6137 where  $f = \mathbf{f}(\mathbf{op})$ .

6138 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6139 • If  $\mathbf{accum} = \mathbf{GrB\_NULL}$ , then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6140 • If  $\mathbf{accum}$  is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6141 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6142 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6143 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$6144 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$6145 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$6146 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

6147 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

6150 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
6151 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
6152 mask which acts as a “write mask”.

- 6153 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
6154 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6155 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6156 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
6157 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
6158 mask are unchanged:

$$6159 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6160 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
6161 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
6162 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
6163 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
6164 sequence.

#### 6165 4.3.8.5 apply: Vector index unary operator variant[Scott: NEW CONTENT]

6166 Computes the transformation of the values of the stored elements of a vector using an index unary  
6167 operator that is a function of the stored value, its location indices, and an user provided scalar  
6168 value. The scalar can be passed either as a non-opaque variable or as a GrB\_Scalar object.

#### 6169 C Syntax

```
6170      GrB_Info GrB_apply(GrB_Vector          w,
6171                        const GrB_Vector    mask,
6172                        const GrB_BinaryOp   accum,
6173                        const GrB_IndexUnaryOp op,
6174                        const GrB_Vector    u,
6175                        <type>              val,
6176                        const GrB_Descriptor desc);
```

```
6177      GrB_Info GrB_apply(GrB_Vector          w,
6178                        const GrB_Vector    mask,
6179                        const GrB_BinaryOp   accum,
6180                        const GrB_IndexUnaryOp op,
6181                        const GrB_Vector    u,
6182                        const GrB_Scalar    s,
6183                        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.

6215                   GrB\_PANIC   Unknown internal error.

6216           GrB\_INVALID\_OBJECT   This is returned in any execution mode whenever one of the  
6217                                   opaque GraphBLAS objects (input or output) is in an invalid  
6218                                   state caused by a previous execution error. Call `GrB_error()` to  
6219                                   access any error messages generated by the implementation.

6220           GrB\_OUT\_OF\_MEMORY   Not enough memory available for operation.

6221   GrB\_UNINITIALIZED\_OBJECT   One or more of the GraphBLAS objects has not been initialized  
6222                                   by a call to `new` (or another constructor).

6223   GrB\_DIMENSION\_MISMATCH   `mask`, `w` and/or `u` dimensions are incompatible.

6224   GrB\_DOMAIN\_MISMATCH   The domains of the various vectors are incompatible with the cor-  
6225                                   responding domains of the accumulation operator or index unary  
6226                                   operator, or the mask's domain is not compatible with `bool` (in  
6227                                   the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6228           GrB\_EMPTY\_OBJECT   The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and  
6229                                   therefore a value cannot be passed to the index unary operator.

## 6230 Description

6231   This variant of `GrB_apply` computes the result of applying an index unary operator to the elements  
6232   of a GraphBLAS vector each composed with the element's index and a scalar constant, `val` or `s`:

$$6233 \quad w = f_i(u, \mathbf{ind}(u), 0, \text{val}) \text{ or } w = f_i(u, \mathbf{ind}(u), 0, s),$$

6234   or if an optional binary accumulation operator ( $\odot$ ) is provided:

$$6235 \quad w = w \odot f_i(u, \mathbf{ind}(u), 0, \text{val}) \text{ or } w = w \odot f_i(u, \mathbf{ind}(u), 0, s).$$

6236   Logically, this operation occurs in three steps:

6237       **Setup**   The internal vectors and mask used in the computation are formed and their domains  
6238                   and dimensions are tested for compatibility.

6239       **Compute**   The indicated computations are carried out.

6240       **Output**   The result is written into the output vector, possibly under control of a mask.

6241   Up to three argument vectors are used in this `GrB_apply` operation:

- 6242   1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6243   2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)

6244 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6245 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)  
6246 are tested for domain compatibility as follows:

- 6247 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
6248 must be from one of the pre-defined types of Table 3.2.
- 6249 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the index unary operator.
- 6250 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
6251 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the index unary operator must be compatible  
6252 with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 6253 4.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the index unary operator.
- 6254 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  
6255 the index unary operator.
- 6256 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  
6257 unary operator.

6258 Two domains are compatible with each other if values from one domain can be cast to values in  
6259 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6260 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6261 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
6262 error listed above is returned.

6263 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
6264 denotes copy):

- 6265 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 6266 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 6267 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 6268 (b) If `mask`  $\neq$  `GrB_NULL`,
    - 6269 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,
    - 6270 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 6271 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 6272 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 6273 4. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).

6274 The internal vectors and masks are checked for dimension compatibility. The following conditions  
6275 must hold:

6276 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

6277 2.  $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$ .

6278 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6279 error listed above is returned.

6280 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6281 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6282 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.

6283 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided ( $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable `val`  
6284 with the same domain as  $\tilde{s}$  and set `val` = `val`( $\tilde{s}$ ).

6285 We are now ready to carry out the apply and any additional associated operations. We describe  
6286 this in terms of two intermediate vectors:

- 6287 •  $\tilde{\mathbf{t}}$ : The vector holding the result from applying the index unary operator to the input vector  
6288  $\tilde{\mathbf{u}}$ .
- 6289 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6290 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$6291 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f_i(\tilde{\mathbf{u}}(i), [i], 0, \text{val})) \mid i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,$$

6292 where  $f_i = \mathbf{f}(\text{op})$ .

6293 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6294 • If `accum` = `GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 6295 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$6296 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6297 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
6298 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 6299 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 6300 \\ 6301 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6302 \\ 6303 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

6304 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

6305 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
6306 using what is called a *standard vector mask and replace*. This is carried out under control of the  
6307 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.

6343 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
6344 stored into the output matrix **C**. The mask dimensions must match those of the  
6345 matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6346 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
6347 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6348 dimensions of **C**), **GrB\_NULL** should be specified.

6349 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
6350 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6351 specified.

6352 **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  
6353 to each element stored in the input matrix, **A**. It is a function of the stored element’s  
6354 value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6355 **A** (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-  
6356 ator.

6357 **val** (IN) An additional scalar value that is passed to the index unary operator.

6358 **s** (IN) An additional GraphBLAS scalar that is passed to the index unary operator.

6359 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
6360 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.

## 6363 Return Values

6364 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
6365 blocking mode, this indicates that the compatibility tests on di-  
6366 mensions and domains for the input arguments passed successfully.  
6367 Either way, output matrix **C** is ready to be used in the next method  
6368 of the sequence.

6369 **GrB\_PANIC** Unknown internal error.

6370 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
6371 GraphBLAS objects (input or output) is in an invalid state caused

6372 by a previous execution error. Call `GrB_error()` to access any error  
 6373 messages generated by the implementation.

6374 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

6375 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 6376 a call to `new` (or another constructor).

6377 **GrB\_DIMENSION\_MISMATCH** `mask`, `w` and/or `u` dimensions are incompatible.

6378 **GrB\_DOMAIN\_MISMATCH** The domains of the various matrices are incompatible with the  
 6379 corresponding domains of the accumulation operator or index unary  
 6380 operator, or the mask's domain is not compatible with `bool` (in the  
 6381 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6382 **GrB\_EMPTY\_OBJECT** The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and  
 6383 therefore a value cannot be passed to the index unary operator.

## 6384 Description

6385 This variant of `GrB_apply` computes the result of applying a index unary operator to the elements  
 6386 of a GraphBLAS matrix each composed with the elements row and column indices, and a scalar  
 6387 constant, `val` or `s`:

$$6388 \quad C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s),$$

6389 or if an optional binary accumulation operator ( $\odot$ ) is provided:

$$6390 \quad C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s).$$

6391 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional  
 6392 indices, respectively.

6393 Logically, this operation occurs in three steps:

6394 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 6395 and dimensions are tested for compatibility.

6396 **Compute** The indicated computations are carried out.

6397 **Output** The result is written into the output matrix, possibly under control of a mask.

6398 Up to three argument matrices are used in the `GrB_apply` operation:

- 6399 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6400 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)

6401 3.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

6402 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)  
6403 are tested for domain compatibility as follows:

- 6404 1. If **Mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\mathbf{Mask})$   
6405 must be from one of the pre-defined types of Table 3.2.
- 6406 2.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator.
- 6407 3. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
6408 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator must be compatible  
6409 with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accumulation operator.
- 6410 4.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the index unary operator.
- 6411 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  
6412 the index unary operator.
- 6413 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  
6414 unary operator.

6415 Two domains are compatible with each other if values from one domain can be cast to values in  
6416 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6417 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6418 any compatibility rule above is violated, execution of **GrB\_apply** ends and the domain mismatch  
6419 error listed above is returned.

6420 From the argument matrices, the internal matrices, **mask**, and index arrays used in the computation  
6421 are formed ( $\leftarrow$  denotes copy):

- 6422 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 6423 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument **Mask** as follows:
  - 6424 (a) If **Mask** = **GrB\_NULL**, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
6425  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 6426 (b) If **Mask**  $\neq$  **GrB\_NULL**,
    - 6427 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
6428  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 6429 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
6430  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 6431 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 6432 3. Matrix  $\tilde{\mathbf{A}}$  is computed from argument **A** as follows:
  - 6433  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc[GrB_INP0].GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$
- 6434 4. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).



6435 The internal matrices and mask are checked for dimension compatibility. The following conditions  
6436 must hold:

- 6437 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 6438 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 6439 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
- 6440 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

6441 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6442 error listed above is returned.

6443 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6444 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6445 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6446 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable  
6447 `val` with the same domain as  $\tilde{s}$  and set `val = val( $\tilde{s}$ )`.

6448 We are now ready to carry out the apply and any additional associated operations. We describe  
6449 this in terms of two intermediate matrices:

- 6450 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the index unary operator to the input matrix  
6451  $\tilde{\mathbf{A}}$ .
- 6452 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6453 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$6454 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f_i(\tilde{\mathbf{A}}(i, j), i, j, \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

6455 where  $f_i = \mathbf{f}(\mathbf{op})$ .

6456 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6457 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6458 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6459 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6460 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6461 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 6462 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6463 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6464 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6465 \end{aligned}$$

6466 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

6468 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 6469 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 6470 mask which acts as a “write mask”.

- 6471 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
 6472 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6473 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6474 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 6475 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 6476 mask are unchanged:

$$6477 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6478 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 6479 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 6480 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
 6481 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 6482 sequence.

#### 6483 4.3.9 select:

6484 Apply a select operator to the stored elements of an object to determine whether or not to keep  
 6485 them.

##### 6486 4.3.9.1 select: Vector variant[Scott: NEW CONTENT]

6487 Apply a select operator (an index unary operator) to the elements of a vector.

#### 6488 C Syntax

```
6489 // scalar value variant
6490 GrB_Info GrB_select(GrB_Vector          w,
6491                    const GrB_Vector      mask,
6492                    const GrB_BinaryOp    accum,
6493                    const GrB_IndexUnaryOp op,
6494                    const GrB_Vector      u,
6495                    <type>                 val,
6496                    const GrB_Descriptor   desc);
6497
6498 // GraphBLAS scalar variant
6499 GrB_Info GrB_select(GrB_Vector          w,
6500                    const GrB_Vector      mask,
```

```

6501         const GrB_BinaryOp      accum,
6502         const GrB_IndexUnaryOp  op,
6503         const GrB_Vector        u,
6504         const GrB_Scalar        s,
6505         const GrB_Descriptor    desc);
6506

```

## 6507 Parameters

6508     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
6509     that may be accumulated with the result of the select operation. On output, this  
6510     vector holds the results of the operation.

6511     **mask** (IN) An optional “write” mask that controls which results from this operation are  
6512     stored into the output vector **w**. The mask dimensions must match those of the  
6513     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6514     of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
6515     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6516     dimensions of **w**), **GrB\_NULL** should be specified.

6517     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
6518     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6519     specified.

6520     **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  
6521     to each element stored in the input vector, **u**. It is a function of the stored element’s  
6522     value, its location index, and a user supplied scalar value (either **s** or **val**).

6523     **u** (IN) The GraphBLAS vector whose elements are passed to the index unary oper-  
6524     ator.

6525     **val** (IN) An additional scalar value that is passed to the index unary operator.

6526     **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must  
6527     not be empty.

6528     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
6529     should be specified. Non-default field/value pairs are listed as follows:

6530

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> .

6531

## 6532 Return Values

6533           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
6534                       blocking mode, this indicates that the compatibility tests on di-  
6535                       mensions and domains for the input arguments passed success-  
6536                       fully. Either way, output vector **w** is ready to be used in the next  
6537                       method of the sequence.

6538           GrB\_PANIC Unknown internal error.

6539           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
6540                       opaque GraphBLAS objects (input or output) is in an invalid  
6541                       state caused by a previous execution error. Call **GrB\_error()** to  
6542                       access any error messages generated by the implementation.

6543           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

6544           GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
6545                       by a call to one of its constructors.

6546           GrB\_DIMENSION\_MISMATCH **mask**, **w** and/or **u** dimensions are incompatible.

6547           GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with the cor-  
6548                       responding domains of the accumulation operator or index unary  
6549                       operator, or the **mask**'s domain is not compatible with **bool** (in  
6550                       the case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

6551           GrB\_EMPTY\_OBJECT The **GrB\_Scalar s** used in the call is empty (**nvals(s) = 0**) and  
6552                       therefore a value cannot be passed to the index unary operator.

## 6553 Description

6554 This variant of **GrB\_select** computes the result of applying a index unary operator to select the  
6555 elements of the input GraphBLAS vector. The operator takes, as input, the value of each stored  
6556 element, along with the element's index and a scalar constant – either **val** or **s**. The corresponding  
6557 element of the input vector is selected (kept) if the function evaluates to **true** when cast to **bool**.  
6558 This acts like a functional mask on the input vector as follows:

$$6559 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle,$$

$$6560 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle.$$

6561 Correspondingly, if a **GrB\_Scalar s**, is provided:

$$6562 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle,$$

$$6563 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle.$$

6564 Logically, this operation occurs in three steps:

6565     **Setup** The internal vectors and mask used in the computation are formed and their domains  
6566             and dimensions are tested for compatibility.

6567     **Compute** The indicated computations are carried out.

6568     **Output** The result is written into the output vector, possibly under control of a mask.

6569 Up to three argument vectors are used in this GrB\_select operation:

- 6570     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$   
6571     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)  
6572     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6573 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)  
6574 are tested for domain compatibility as follows:

- 6575     1. If **mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\mathbf{mask})$   
6576         must be from one of the pre-defined types of Table 3.2.  
6577     2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .  
6578     3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
6579         of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accu-  
6580         mulation operator.  
6581     4.  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator must be from one of the pre-defined types of Table 3.2;  
6582         i.e., castable to **bool**.  
6583     5.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the index unary operator.  
6584     6.  $\mathbf{D}(\mathbf{val})$  or  $\mathbf{D}(\mathbf{s})$ , depending on the signature of the method, must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$   
6585         of the index unary operator.

6586 Two domains are compatible with each other if values from one domain can be cast to values in  
6587 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6588 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6589 any compatibility rule above is violated, execution of GrB\_select ends and the domain mismatch  
6590 error listed above is returned.

6591 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
6592 denotes copy):

- 6593     1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .  
6594     2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument **mask** as follows:

6595 (a) If  $\text{mask} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .  
6596 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,  
6597 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,  
6598 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .  
6599 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .  
6600 3. Vector  $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$ .  
6601 4. Scalar  $\widetilde{s} \leftarrow s$  (GrB\_Scalar version only).

6602 The internal vectors and masks are checked for dimension compatibility. The following conditions  
6603 must hold:

- 6604 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 6605 2.  $\text{size}(\widetilde{\mathbf{u}}) = \text{size}(\widetilde{\mathbf{w}})$ .

6606 If any compatibility rule above is violated, execution of  $\text{GrB\_select}$  ends and the dimension mismatch  
6607 error listed above is returned.

6608 From this point forward, in  $\text{GrB\_NONBLOCKING}$  mode, the method can optionally exit with  
6609  $\text{GrB\_SUCCESS}$  return code and defer any computation and/or execution error codes.

6610 If an empty  $\text{GrB\_Scalar } \widetilde{s}$  is provided (i.e.,  $\text{nvals}(\widetilde{s}) = 0$ ), the method returns with code  $\text{GrB\_EMPTY\_OBJECT}$ .  
6611 If a non-empty  $\text{GrB\_Scalar } \widetilde{s}$ , is provided (i.e.,  $\text{nvals}(\widetilde{s}) = 1$ ), we then create an internal variable  
6612  $\text{val}$  with the same domain as  $\widetilde{s}$  and set  $\text{val} = \text{val}(\widetilde{s})$ .

6613 We are now ready to carry out the  $\text{select}$  and any additional associated operations. We describe  
6614 this in terms of two intermediate vectors:

- 6615 •  $\widetilde{\mathbf{t}}$ : The vector holding the result from applying the index unary operator to the input vector  
6616  $\widetilde{\mathbf{u}}$ .
- 6617 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6618 The intermediate vector,  $\widetilde{\mathbf{t}}$ , is created as follows:

$$6619 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{u}}), \{(i, \widetilde{\mathbf{u}}(i), : i \in \text{ind}(\widetilde{\mathbf{u}}) \wedge (\text{bool})f_i(\widetilde{\mathbf{u}}(i), i, 0, \text{val}) = \text{true})\} \rangle,$$

6620 where  $f_i = \mathbf{f}(\text{op})$ .

6621 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6622 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$ .
- 6623 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{z}}$  is defined as

$$6624 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

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);
```

```

6660 // GraphBLAS scalar variant
6661 GrB_Info GrB_select(GrB_Matrix          C,
6662                    const GrB_Matrix     Mask,
6663                    const GrB_BinaryOp    accum,
6664                    const GrB_IndexUnaryOp op,
6665                    const GrB_Matrix      A,
6666                    const GrB_Scalar      s,
6667                    const GrB_Descriptor   desc);

```

## 6668 Parameters

6669 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
6670 that may be accumulated with the result of the select operation. On output, the  
6671 matrix holds the results of the operation.

6672 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
6673 stored into the output matrix **C**. The mask dimensions must match those of the  
6674 matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6675 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
6676 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6677 dimensions of **C**), **GrB\_NULL** should be specified.

6678 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
6679 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6680 specified.

6681 **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  
6682 to each element stored in the input matrix, **A**. It is a function of the stored element’s  
6683 value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6684 **A** (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-  
6685 ator.

6686 **val** (IN) An additional scalar value that is passed to the index unary operator.

6687 **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must  
6688 not be empty.

6689 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
6690 should be specified. Non-default field/value pairs are listed as follows:

6691



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:

6721  $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$ , or  
 6722  $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$ .

6723 Correspondingly, if a GrB\_Scalar,  $s$ , is provided:

6724  $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$ , or  
 6725  $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$ .

6726 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional  
 6727 indices, respectively.

6728 Logically, this operation occurs in three steps:

6729 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 6730 and dimensions are tested for compatibility.

6731 **Compute** The indicated computations are carried out.

6732 **Output** The result is written into the output matrix, possibly under control of a mask.

6733 Up to three argument matrices are used in the GrB\_select operation:

- 6734 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6735 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)
- 6736 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6737 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)  
 6738 are tested for domain compatibility as follows:

- 6739 1. If **Mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{Mask})$   
 6740 must be from one of the pre-defined types of Table 3.2.
- 6741 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$ .
- 6742 3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 6743 of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 6744 mulation operator.
- 6745 4.  $\mathbf{D}_{out}(\text{op})$  of the index unary operator must be from one of the pre-defined types of Table 3.2;  
 6746 i.e., castable to **bool**.
- 6747 5.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the index unary operator.
- 6748 6.  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(s)$ , depending on the signature of the method, must be compatible with  $\mathbf{D}_{in_2}(\text{op})$   
 6749 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:

- 6784 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the index unary operator to the input matrix  
6785  $\tilde{\mathbf{A}}$ .
- 6786 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6787 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$6788 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \\ \{(i, j, \tilde{\mathbf{A}}(i, j) : i, j \in \mathbf{ind}(\tilde{\mathbf{A}}) \wedge (\text{bool})f_i(\tilde{\mathbf{A}}(i, j), i, j, \text{val}) = \text{true})\},$$

6789 where  $f_i = \mathbf{f}(\text{op})$ .

6790 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6791 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6792 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6793 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\}\rangle.$$

6794 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6795 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$6796 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6797 \\ 6798 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6799 \\ 6800 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

6801 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

6802 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
6803 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
6804 mask which acts as a “write mask”.

- 6805 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
6806 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6807 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6808 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
6809 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
6810 mask are unchanged:

$$6811 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6812 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
6813 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
6814 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
6815 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
6816 sequence.

### 6817 4.3.10 reduce: Perform a reduction across the elements of an object

6818 Computes the reduction of the values of the elements of a vector or matrix.

#### 6819 4.3.10.1 reduce: Standard matrix to vector variant

6820 This performs a reduction across rows of a matrix to produce a vector. If reduction down columns  
6821 is desired, the input matrix should be transposed using the descriptor.

### 6822 C Syntax

```
6823     GrB_Info GrB_reduce(GrB_Vector      w,  
6824                        const GrB_Vector mask,  
6825                        const GrB_BinaryOp accum,  
6826                        const GrB_Monoid op,  
6827                        const GrB_Matrix A,  
6828                        const GrB_Descriptor desc);  
6829  
6830     GrB_Info GrB_reduce(GrB_Vector      w,  
6831                        const GrB_Vector mask,  
6832                        const GrB_BinaryOp accum,  
6833                        const GrB_BinaryOp op,  
6834                        const GrB_Matrix A,  
6835                        const GrB_Descriptor desc);
```

### 6836 Parameters

6837 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
6838 that may be accumulated with the result of the reduction operation. On output,  
6839 this vector holds the results of the operation.

6840 **mask** (IN) An optional “write” mask that controls which results from this operation are  
6841 stored into the output vector **w**. The mask dimensions must match those of the  
6842 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6843 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
6844 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6845 dimensions of **w**), **GrB\_NULL** should be specified.

6846 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
6847 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6848 specified.

6849 **op** (IN) The monoid or binary operator used in the element-wise reduction operation.  
6850 Depending on which type is passed, the following defines the binary operator with  
6851 one domain,  $F_b = \langle D, D, D, \oplus \rangle$ , that is used:

6852 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .  
6853 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  
6854 monoid is ignored.

6855 If  $\text{op}$  is a `GrB_BinaryOp`, then all its domains must be the same. Furthermore, in  
6856 both cases  $\odot(\text{op})$  must be commutative and associative. Otherwise, the outcome  
6857 of the operation is undefined.

6858 **A** (IN) The GraphBLAS matrix on which reduction will be performed.

6859 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
6860 should be specified. Non-default field/value pairs are listed as follows:  
6861

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.

## 6863 Return Values

6864 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
6865 blocking mode, this indicates that the compatibility tests on di-  
6866 mensions and domains for the input arguments passed successfully.  
6867 Either way, output vector w is ready to be used in the next method  
6868 of the sequence.

6869 **GrB\_PANIC** Unknown internal error.

6870 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
6871 GraphBLAS objects (input or output) is in an invalid state caused  
6872 by a previous execution error. Call `GrB_error()` to access any error  
6873 messages generated by the implementation.

6874 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

6875 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
6876 a call to `new` (or `dup` for vector parameters).

6877 **GrB\_DIMENSION\_MISMATCH** mask, w and/or u dimensions are incompatible.

6878 **GrB\_DOMAIN\_MISMATCH** Either the domains of the various vectors and matrices are incom-  
6879 patible with the corresponding domains of the accumulation oper-  
6880 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).

## 6884 Description

6885 This variant of `GrB_reduce` computes the result of performing a reduction across each of the rows  
 6886 of an input matrix:  $w(i) = \bigoplus A(i, :) \forall i$ ; or, if an optional binary accumulation operator is provided,  
 6887  $w(i) = w(i) \odot (\bigoplus A(i, :)) \forall i$ , where  $\bigoplus = \odot(F_b)$  and  $\odot = \odot(\text{accum})$ .

6888 Logically, this operation occurs in three steps:

6889     **Setup** The internal vector, matrix and mask used in the computation are formed and their  
 6890 domains and dimensions are tested for compatibility.

6891     **Compute** The indicated computations are carried out.

6892     **Output** The result is written into the output vector, possibly under control of a mask.

6893 Up to two vector and one matrix argument are used in this `GrB_reduce` operation:

- 6894 1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6895 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 6896 3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6897 The argument vector, matrix, reduction operator and accumulation operator (if provided) are tested  
 6898 for domain compatibility as follows:

- 6899 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
 6900 must be from one of the pre-defined types of Table 3.2.
- 6901 2.  $\mathbf{D}(w)$  must be compatible with the domain of the reduction binary operator,  $\mathbf{D}(F_b)$ .
- 6902 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 6903 of the accumulation operator and  $\mathbf{D}(F_b)$ , must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 6904 mulation operator.
- 6905 4.  $\mathbf{D}(A)$  must be compatible with the domain of the binary reduction operator,  $\mathbf{D}(F_b)$ .

6906 Two domains are compatible with each other if values from one domain can be cast to values in  
 6907 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 6908 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 6909 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch  
 6910 error listed above is returned.

6911 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
 6912 denotes copy):

- 6913 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 6914 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
- 6915 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
- 6916 (b) If `mask  $\neq$  GrB_NULL`,
- 6917 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
- 6918 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
- 6919 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 6920 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

6921 The internal vectors and masks are checked for dimension compatibility. The following conditions  
 6922 must hold:

- 6923 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 6924 2.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .

6925 If any compatibility rule above is violated, execution of `GrB_reduce` ends and the dimension mis-  
 6926 match error listed above is returned.

6927 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 6928 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6929 We carry out the reduce and any additional associated operations. We describe this in terms of  
 6930 two intermediate vectors:

- 6931 •  $\tilde{\mathbf{t}}$ : The vector holding the result from reducing along the rows of input matrix  $\tilde{\mathbf{A}}$ .
- 6932 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6933 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$6934 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, t_i) : \mathbf{ind}(\mathbf{A}(i, :)) \neq \emptyset\} \rangle.$$

6935 The value of each of its elements is computed by

$$6936 \quad t_i = \bigoplus_{j \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :))} \tilde{\mathbf{A}}(i, j),$$

6937 where  $\bigoplus = \odot(F_b)$ .

6938 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6939 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .



6940 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$6941 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6942 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
6943 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 6944 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 6945 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 6946 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

6949 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

6950 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
6951 using what is called a *standard vector mask and replace*. This is carried out under control of the  
6952 mask which acts as a “write mask”.

6953 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
6954 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$6955 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

6956 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
6957 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
6958 mask are unchanged:

$$6959 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

6960 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
6961 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
6962 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but  
6963 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
6964 sequence.

#### 6965 4.3.10.2 reduce: Vector-scalar variant[Scott: NEW CONTENT]

6966 Reduce all stored values into a single scalar.

#### 6967 C Syntax

```
6968 // scalar value + monoid (only)
6969 GrB_Info GrB_reduce(<type>          *val,
6970                      const GrB_BinaryOp accum,
6971                      const GrB_Monoid  op,
6972                      const GrB_Vector  u,
```

```

6973             const GrB_Descriptor desc);
6974
6975 // GraphBLAS Scalar + monoid
6976 GrB_Info GrB_reduce(GrB_Scalar      s,
6977                    const GrB_BinaryOp accum,
6978                    const GrB_Monoid op,
6979                    const GrB_Vector u,
6980                    const GrB_Descriptor desc);
6981
6982 // GraphBLAS Scalar + binary operator
6983 GrB_Info GrB_reduce(GrB_Scalar      s,
6984                    const GrB_BinaryOp accum,
6985                    const GrB_BinaryOp op,
6986                    const GrB_Vector u,
6987                    const GrB_Descriptor desc);

```

## 6988 Parameters

6989 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides  
6990 a value that may be accumulated (optionally) with the result of the reduction  
6991 operation. On output, this scalar holds the results of the operation.

6992 **accum** (IN) An optional binary operator used for accumulating entries into an exist-  
6993 ing scalar (**s** or **val**) value. If assignment rather than accumulation is desired,  
6994 **GrB\_NULL** should be specified.

6995 **op** (IN) The monoid ( $M = \langle D, \oplus, 0 \rangle$ ) or binary operator ( $F_b = \langle D, D, D, \oplus \rangle$ ) used in  
6996 the reduction operation. The  $\oplus$  operator must be commutative and associative;  
6997 otherwise, the outcome of the operation is undefined.

6998 **u** (IN) The GraphBLAS vector on which reduction will be performed.

6999 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
7000 should be specified. Non-default field/value pairs are listed as follows:

7002 Param	Field	Value	Description
------------	-------	-------	-------------

7003 *Note:* This argument is defined for consistency with the other GraphBLAS opera-  
7004 tions. There are currently no non-default field/value pairs that can be set for this  
7005 operation.

## 7006 Return Values

7007 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
7008 cessfully, and the output scalar (**s** or **val**) is ready to be used in the  
7009 next method of the sequence.

7010                   GrB\_PANIC Unknown internal error.

7011           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
7012                   GraphBLAS objects (input or output) is in an invalid state caused  
7013                   by a previous execution error. Call GrB\_error() to access any error  
7014                   messages generated by the implementation.

7015           GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

7016 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
7017                   a call to a respective constructor.

7018           GrB\_NULL\_POINTER val pointer is NULL.

7019           GrB\_DOMAIN\_MISMATCH The domains of input and output arguments are incompatible with  
7020                   the corresponding domains of the accumulation operator, or reduce  
7021                   operator.

## 7022 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):

$$\text{val} = \begin{cases} \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i), & \text{or} \\ \text{val} \odot \left[ \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7023 where  $\bigoplus = \odot(\text{op})$  and  $\odot = \odot(\text{accum})$ .

7024 Logically, this operation occurs in three steps:

7025       **Setup** The internal vector used in the computation is formed and its domain is tested for  
7026                   compatibility.

7027       **Compute** The indicated computations are carried out.

7028       **Output** The result is written into the output scalar.

7029 One vector argument is used in this GrB\_reduce operation:

- 7030 1.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

7031 The output scalar, argument vector, reduction operator and accumulation operator (if provided)  
7032 are tested for domain compatibility as follows:

- 7033 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  
7034  $\mathbf{D}_{in_1}(\text{op})$  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  $\tilde{\mathbf{u}} \leftarrow \text{u}$ .
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

$$\text{val} \leftarrow \begin{cases} t, & \text{when } \text{accum} = \text{GrB\_NULL, or} \\ \text{val} \odot t, & \text{otherwise;} \end{cases}$$

7062 In both GrB\_BLOCKING and GrB\_NONBLOCKING modes, the method exits with return value  
 7063 GrB\_SUCCESS and the new contents of the output scalar is as defined above.

#### 7064 4.3.10.3 reduce: Matrix-scalar variant[Scott: NEW CONTENT]

7065 Reduce all stored values into a single scalar.

#### 7066 C Syntax

```

7067 // scalar value + monoid (only)
7068 GrB_Info GrB_reduce(<type>          *val,
7069                   const GrB_BinaryOp accum,
7070                   const GrB_Monoid   op,
7071                   const GrB_Matrix   A,
7072                   const GrB_Descriptor desc);
7073
7074 // GraphBLAS Scalar + monoid
7075 GrB_Info GrB_reduce(GrB_Scalar      s,
7076                   const GrB_BinaryOp accum,
7077                   const GrB_Monoid   op,
7078                   const GrB_Matrix   A,
7079                   const GrB_Descriptor desc);
7080
7081 // GraphBLAS Scalar + binary operator
7082 GrB_Info GrB_reduce(GrB_Scalar      s,
7083                   const GrB_BinaryOp accum,
7084                   const GrB_BinaryOp op,
7085                   const GrB_Matrix   A,
7086                   const GrB_Descriptor desc);

```

#### 7087 Parameters

7088 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides  
 7089 a value that may be accumulated (optionally) with the result of the reduction  
 7090 operation. On output, this scalar holds the results of the operation.

7091 **accum** (IN) An optional binary operator used for accumulating entries into existing (**s** or  
 7092 **val**) value. If assignment rather than accumulation is desired, GrB\_NULL should  
 7093 be specified.

7094 **op** (IN) The monoid ( $M = \langle D, \oplus, 0 \rangle$ ) or binary operator ( $F_b = \langle D, D, D, \oplus \rangle$ ) used in  
 7095 the reduction operation. The  $\oplus$  operator must be commutative and associative;  
 7096 otherwise, the outcome of the operation is undefined.

7097 **A** (IN) The GraphBLAS matrix on which the reduction will be performed.

7098 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
7099 should be specified. Non-default field/value pairs are listed as follows:  
7100

7101	Param	Field	Value	Description
------	-------	-------	-------	-------------

7102 *Note:* This argument is defined for consistency with the other GraphBLAS opera-  
7103 tions. There are currently no non-default field/value pairs that can be set for this  
7104 operation.

## 7105 Return Values

7106 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
7107 cessfully, and the output scalar (s or val) is ready to be used in the  
7108 next method of the sequence.

7109 GrB\_PANIC Unknown internal error.

7110 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
7111 GraphBLAS objects (input or output) is in an invalid state caused  
7112 by a previous execution error. Call GrB\_error() to access any error  
7113 messages generated by the implementation.

7114 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

7115 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
7116 a call to a respective constructor.

7117 GrB\_NULL\_POINTER val pointer is NULL.

7118 GrB\_DOMAIN\_MISMATCH The domains of input and output arguments are incompatible with  
7119 the corresponding domains of the accumulation operator, or reduce  
7120 operator.

## 7121 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):

$$\text{val} = \begin{cases} \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j), & \text{or} \\ \text{val} \odot \left[ \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7122 where  $\bigoplus = \odot(\text{op})$  and  $\odot = \odot(\text{accum})$ .

7123 Logically, this operation occurs in three steps:

7124       **Setup** The internal matrix used in the computation is formed and its domain is tested for  
 7125       compatibility.

7126       **Compute** The indicated computations are carried out.

7127       **Output** The result is written into the output scalar.

7128       One matrix argument is used in this GrB\_reduce operation:

7129       1.  $A = \langle \mathbf{D}(A), \mathbf{size}(A), \mathbf{L}(A) = \{(i, j, A_{i,j})\} \rangle$

7130       The output scalar, argument matrix, reduction operator and accumulation operator (if provided)  
 7131       are tested for domain compatibility as follows:

7132       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  
 7133        $\mathbf{D}_{in_1}(\text{op})$  and  $\mathbf{D}_{in_2}(\text{op})$  from  $F_b$ ).

7134       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  
 7135        $\mathbf{D}_{out}(\text{accum})$  of the accumulation operator, and  $\mathbf{D}(\text{op})$  from  $M$  (or  $\mathbf{D}_{out}(\text{op})$  from  $F_b$ ) must  
 7136       be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.

7137       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  $F_b$ ).

7138       Two domains are compatible with each other if values from one domain can be cast to values in  
 7139       the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 7140       compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 7141       any compatibility rule above is violated, execution of GrB\_reduce ends and the domain mismatch  
 7142       error listed above is returned.

7143       The number of values stored in the input,  $A$ , is checked. If there are no stored values in  $A$ , then  
 7144       one of the following occurs depending on the output variant:

$$7145 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if accum = GrB\_NULL,} \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7146       or

$$7147 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if accum = GrB\_NULL,} \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7148       where  $\mathbf{0}(\text{op})$  is the identity of the monoid. The operation returns immediately with GrB\_SUCCESS.

7149       For all other cases, the internal matrix and scalar used in the computation is formed ( $\leftarrow$  denotes  
 7150       copy):

7151       1. Matrix  $\tilde{A} \leftarrow A$ .

7152       2. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).

7153 We are now ready to carry out the reduce and any additional associated operations. An intermediate  
 7154 scalar result  $t$  is computed as follows:

$$7155 \quad t = \bigoplus_{(i,j) \in \text{ind}(\tilde{\mathbf{A}})} \tilde{\mathbf{A}}(i,j),$$

7156 where  $\oplus = \odot(\text{op})$ .

7157 The final reduction value is computed as follows:

$$7158 \quad \mathbf{L}(\mathbf{s}) \leftarrow \begin{cases} \{t\}, & \text{when accum} = \text{GrB\_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\mathbf{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7159 or

$$7160 \quad \mathbf{val} \leftarrow \begin{cases} t, & \text{when accum} = \text{GrB\_NULL, or} \\ \mathbf{val} \odot t, & \text{otherwise;} \end{cases}$$

7161 In both GrB\_BLOCKING and GrB\_NONBLOCKING modes, the method exits with return value  
 7162 GrB\_SUCCESS and the new contents of the output scalar is as defined above.

#### 7163 4.3.11 transpose: Transpose rows and columns of a matrix

7164 This version computes a new matrix that is the transpose of the source matrix.

#### 7165 C Syntax

```
7166      GrB_Info GrB_transpose(GrB_Matrix      C,
7167                           const GrB_Matrix Mask,
7168                           const GrB_BinaryOp accum,
7169                           const GrB_Matrix A,
7170                           const GrB_Descriptor desc);
```

#### 7171 Parameters

7172 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
 7173 that may be accumulated with the result of the transpose operation. On output,  
 7174 the matrix holds the results of the operation.

7175 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
 7176 stored into the output matrix C. The mask dimensions must match those of the  
 7177 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
 7178 of the Mask matrix must be of type bool or any of the predefined “built-in” types  
 7179 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
 7180 dimensions of C), GrB\_NULL should be specified.



7181       **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
7182       entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
7183       specified.

7184       **A** (IN) The GraphBLAS matrix on which transposition will be performed.

7185       **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
7186       should be specified. Non-default field/value pairs are listed as follows:

7187

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.

7188

## 7189   **Return Values**

7190       **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
7191       blocking mode, this indicates that the compatibility tests on di-  
7192       mensions and domains for the input arguments passed successfully.  
7193       Either way, output matrix **C** is ready to be used in the next method  
7194       of the sequence.

7195       **GrB\_PANIC** Unknown internal error.

7196       **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
7197       GraphBLAS objects (input or output) is in an invalid state caused  
7198       by a previous execution error. Call **GrB\_error()** to access any error  
7199       messages generated by the implementation.

7200       **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

7201       **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
7202       a call to **new** (or **Matrix\_dup** for matrix parameters).

7203       **GrB\_DIMENSION\_MISMATCH** **mask**, **C** and/or **A** dimensions are incompatible.

7204       **GrB\_DOMAIN\_MISMATCH** The domains of the various matrices are incompatible with the cor-  
7205       responding domains of the accumulation operator, or the mask's do-  
7206       main is not compatible with **bool** (in the case where **desc[GrB\_MASK].GrB\_STRUCTURE**  
7207       is not set).

## 7208 Description

7209 GrB\_transpose computes the result of performing a transpose of the input matrix:  $C = A^T$ ; or, if an  
 7210 optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot A^T$ . We note that the input matrix  
 7211 A can itself be optionally transposed before the operation, which would cause either an assignment  
 7212 from A to C or an accumulation of A into C.

7213 Logically, this operation occurs in three steps:

7214     **Setup** The internal matrix and mask used in the computation are formed and their domains  
 7215             and dimensions are tested for compatibility.

7216     **Compute** The indicated computations are carried out.

7217     **Output** The result is written into the output matrix, possibly under control of a mask.

7218 Up to three matrix arguments are used in this GrB\_transpose operation:

- 7219     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7220     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)
- 7221     3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

7222 The argument matrices and accumulation operator (if provided) are tested for domain compatibility  
 7223 as follows:

- 7224     1. If Mask is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{Mask})$   
 7225         must be from one of the pre-defined types of Table 3.2.
- 7226     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$  of the input matrix.
- 7227     3. If accum is not GrB\_NULL, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 7228         of the accumulation operator and  $\mathbf{D}(A)$  of the input matrix must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
 7229         of the accumulation operator.

7230 Two domains are compatible with each other if values from one domain can be cast to values in  
 7231 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 7232 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 7233 any compatibility rule above is violated, execution of GrB\_transpose ends and the domain mismatch  
 7234 error listed above is returned.

7235 From the argument matrices, the internal matrices and mask used in the computation are formed  
 7236 ( $\leftarrow$  denotes copy):

- 7237     1. Matrix  $\tilde{C} \leftarrow C$ .
- 7238     2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument Mask as follows:

- 7239 (a) If  $\text{Mask} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
7240  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
- 7241 (b) If  $\text{Mask} \neq \text{GrB\_NULL}$ ,
- 7242 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
7243  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
- 7244 ii. Otherwise,  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
7245  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
- 7246 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{M}} \leftarrow \neg \widetilde{\mathbf{M}}$ .
- 7247 3. Matrix  $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

7248 The internal matrices and masks are checked for dimension compatibility. The following conditions  
7249 must hold:

- 7250 1.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$ .
- 7251 2.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$ .
- 7252 3.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{A}})$ .
- 7253 4.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$ .

7254 If any compatibility rule above is violated, execution of `GrB_transpose` ends and the dimension  
7255 mismatch error listed above is returned.

7256 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
7257 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

7258 We are now ready to carry out the matrix transposition and any additional associated operations.  
7259 We describe this in terms of two intermediate matrices:

- 7260 •  $\widetilde{\mathbf{T}}$ : The matrix holding the transpose of  $\widetilde{\mathbf{A}}$ .
- 7261 •  $\widetilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

7262 The intermediate matrix

$$7263 \quad \widetilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \{(j, i, A_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{A}})\} \rangle$$

7264 is created.

7265 The intermediate matrix  $\widetilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 7266 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$ .
- 7267 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{Z}}$  is defined as

$$7268 \quad \widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\} \rangle.$$

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);
```

```

7303     GrB_Info GrB_kronecker(GrB_Matrix      C,
7304                           const GrB_Matrix Mask,
7305                           const GrB_BinaryOp accum,
7306                           const GrB_Monoid  op,
7307                           const GrB_Matrix A,
7308                           const GrB_Matrix B,
7309                           const GrB_Descriptor desc);
7310
7311     GrB_Info GrB_kronecker(GrB_Matrix      C,
7312                           const GrB_Matrix Mask,
7313                           const GrB_BinaryOp accum,
7314                           const GrB_BinaryOp op,
7315                           const GrB_Matrix A,
7316                           const GrB_Matrix B,
7317                           const GrB_Descriptor desc);

```

## 7318 Parameters

7319 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
7320 that may be accumulated with the result of the Kronecker product. On output,  
7321 the matrix holds the results of the operation.

7322 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
7323 stored into the output matrix C. The mask dimensions must match those of the  
7324 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
7325 of the Mask matrix must be of type bool or any of the predefined “built-in” types  
7326 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
7327 dimensions of C), GrB\_NULL should be specified.

7328 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
7329 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
7330 specified.

7331 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”  
7332 operation. Depending on which type is passed, the following defines the binary  
7333 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

7334 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

7335 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-  
7336 nored.

7337 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  
7338 is ignored.

7339 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
7340 product.

7341 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
7342 product.

7343 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
7344 should be specified. Non-default field/value pairs are listed as follows:  
7345

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.

## 7347 Return Values

7348 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
7349 blocking mode, this indicates that the compatibility tests on di-  
7350 mensions and domains for the input arguments passed successfully.  
7351 Either way, output matrix C is ready to be used in the next method  
7352 of the sequence.

7353 GrB\_PANIC Unknown internal error.

7354 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
7355 GraphBLAS objects (input or output) is in an invalid state caused  
7356 by a previous execution error. Call GrB\_error() to access any error  
7357 messages generated by the implementation.

7358 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

7359 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
7360 a call to new (or Matrix\_dup for matrix parameters).

7361 GrB\_DIMENSION\_MISMATCH Mask and/or matrix dimensions are incompatible.

7362 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
7363 corresponding domains of the binary operator (op) or accumulation  
7364 operator, or the mask's domain is not compatible with bool (in the  
7365 case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

## 7366 Description

7367 GrB\_kronecker computes the Kronecker product  $C = A \otimes B$  or, if an optional binary accumulation  
7368 operator ( $\odot$ ) is provided,  $C = C \odot (A \otimes B)$  (where matrices A and B can be optionally transposed).

7369 The Kronecker product is defined as follows:

7370

$$7371 \quad C = A \otimes B = \begin{bmatrix} A_{0,0} \otimes B & A_{0,1} \otimes B & \dots & A_{0,n_A-1} \otimes B \\ A_{1,0} \otimes B & A_{1,1} \otimes B & \dots & A_{1,n_A-1} \otimes B \\ \vdots & \vdots & \ddots & \vdots \\ A_{m_A-1,0} \otimes B & A_{m_A-1,1} \otimes B & \dots & A_{m_A-1,n_A-1} \otimes B \end{bmatrix}$$

7372 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  
7373 Kronecker product are defined as

$$7374 \quad C(i_A m_B + i_B, j_A n_B + j_B) = A_{i_A, j_A} \otimes B_{i_B, j_B},$$

7375 where  $\otimes$  is the multiplicative operator specified by the **op** parameter.

7376 Logically, this operation occurs in three steps:

7377 **Setup** The internal matrices and mask used in the computation are formed and their domains  
7378 and dimensions are tested for compatibility.

7379 **Compute** The indicated computations are carried out.

7380 **Output** The result is written into the output matrix, possibly under control of a mask.

7381 Up to four argument matrices are used in the **GrB\_kronecker** operation:

- 7382 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7383 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)
- 7384 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 7385 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

7386 The argument matrices, the "product" operator (**op**), and the accumulation operator (if provided)  
7387 are tested for domain compatibility as follows:

- 7388 1. If **Mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\text{Mask})$   
7389 must be from one of the pre-defined types of Table 3.2.
- 7390 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$ .
- 7391 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$ .
- 7392 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 7393 5. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
7394 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of **op** must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
7395 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.



7429 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}) \times \mathbf{nrows}(\tilde{\mathbf{B}}), \mathbf{ncols}(\tilde{\mathbf{A}}) \times \mathbf{ncols}(\tilde{\mathbf{B}}), \{(i, j, T_{ij}) \text{ where } i =$   
 7430  $i_A \cdot m_B + i_B, j = j_A \cdot n_B + j_B, \forall (i_A, j_A) = \mathbf{ind}(\tilde{\mathbf{A}}), (i_B, j_B) = \mathbf{ind}(\tilde{\mathbf{B}})\}$  is created. The value of  
 7431 each of its elements is computed by

$$7432 \quad T_{i_A \cdot m_B + i_B, j_A \cdot n_B + j_B} = \tilde{\mathbf{A}}(i_A, j_A) \otimes \tilde{\mathbf{B}}(i_B, j_B),$$

7433 where  $\otimes$  is the multiplicative operator specified by the `op` parameter.

7434 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 7435 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 7436 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$7437 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

7438 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 7439 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$7440 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$7441 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$7442 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

7443 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

7446 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 7447 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 7448 mask which acts as a “write mask”.

- 7449 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
 7450 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$7451 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7452 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 7453 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 7454 mask are unchanged:

$$7455 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7456 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
 7457 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
 7458 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
 7459 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 7460 sequence. s



## Chapter 5

# Nonpolymorphic interface[Scott: NEW CONTENT]

Each polymorphic GraphBLAS method (those with multiple parameter signatures under the same name) has a corresponding set of long-name forms that are specific to each parameter signature. That is show in Tables 5.1 through 5.11.

Table 5.1: Long-name, nonpolymorphic form of GraphBLAS methods.

Polymorphic signature	Nonpolymorphic signature
GrB_Monoid_new(GrB_Monoid*,...,bool)	GrB_Monoid_new_BOOL(GrB_Monoid*,GrB_BinaryOp,bool)
GrB_Monoid_new(GrB_Monoid*,...,int8_t)	GrB_Monoid_new_INT8(GrB_Monoid*,GrB_BinaryOp,int8_t)
GrB_Monoid_new(GrB_Monoid*,...,uint8_t)	GrB_Monoid_new_UINT8(GrB_Monoid*,GrB_BinaryOp,uint8_t)
GrB_Monoid_new(GrB_Monoid*,...,int16_t)	GrB_Monoid_new_INT16(GrB_Monoid*,GrB_BinaryOp,int16_t)
GrB_Monoid_new(GrB_Monoid*,...,uint16_t)	GrB_Monoid_new_UINT16(GrB_Monoid*,GrB_BinaryOp,uint16_t)
GrB_Monoid_new(GrB_Monoid*,...,int32_t)	GrB_Monoid_new_INT32(GrB_Monoid*,GrB_BinaryOp,int32_t)
GrB_Monoid_new(GrB_Monoid*,...,uint32_t)	GrB_Monoid_new_UINT32(GrB_Monoid*,GrB_BinaryOp,uint32_t)
GrB_Monoid_new(GrB_Monoid*,...,int64_t)	GrB_Monoid_new_INT64(GrB_Monoid*,GrB_BinaryOp,int64_t)
GrB_Monoid_new(GrB_Monoid*,...,uint64_t)	GrB_Monoid_new_UINT64(GrB_Monoid*,GrB_BinaryOp,uint64_t)
GrB_Monoid_new(GrB_Monoid*,...,float)	GrB_Monoid_new_FP32(GrB_Monoid*,GrB_BinaryOp,float)
GrB_Monoid_new(GrB_Monoid*,...,double)	GrB_Monoid_new_FP64(GrB_Monoid*,GrB_BinaryOp,double)
GrB_Monoid_new(GrB_Monoid*,...,other)	GrB_Monoid_new_UDT(GrB_Monoid*,GrB_BinaryOp,void*)

Table 5.2: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Scalar_setElement(..., bool,...)	GrB_Scalar_setElement_BOOL(..., bool,...)
GrB_Scalar_setElement(..., int8_t,...)	GrB_Scalar_setElement_INT8(..., int8_t,...)
GrB_Scalar_setElement(..., uint8_t,...)	GrB_Scalar_setElement_UINT8(..., uint8_t,...)
GrB_Scalar_setElement(..., int16_t,...)	GrB_Scalar_setElement_INT16(..., int16_t,...)
GrB_Scalar_setElement(..., uint16_t,...)	GrB_Scalar_setElement_UINT16(..., uint16_t,...)
GrB_Scalar_setElement(..., int32_t,...)	GrB_Scalar_setElement_INT32(..., int32_t,...)
GrB_Scalar_setElement(..., uint32_t,...)	GrB_Scalar_setElement_UINT32(..., uint32_t,...)
GrB_Scalar_setElement(..., int64_t,...)	GrB_Scalar_setElement_INT64(..., int64_t,...)
GrB_Scalar_setElement(..., uint64_t,...)	GrB_Scalar_setElement_UINT64(..., uint64_t,...)
GrB_Scalar_setElement(..., float,...)	GrB_Scalar_setElement_FP32(..., float,...)
GrB_Scalar_setElement(..., double,...)	GrB_Scalar_setElement_FP64(..., double,...)
GrB_Scalar_setElement(..., <i>other</i> ,...)	GrB_Scalar_setElement_UDT(..., const void*,...)
GrB_Scalar_extractElement(bool*,...)	GrB_Scalar_extractElement_BOOL(bool*,...)
GrB_Scalar_extractElement(int8_t*,...)	GrB_Scalar_extractElement_INT8(int8_t*,...)
GrB_Scalar_extractElement(uint8_t*,...)	GrB_Scalar_extractElement_UINT8(uint8_t*,...)
GrB_Scalar_extractElement(int16_t*,...)	GrB_Scalar_extractElement_INT16(int16_t*,...)
GrB_Scalar_extractElement(uint16_t*,...)	GrB_Scalar_extractElement_UINT16(uint16_t*,...)
GrB_Scalar_extractElement(int32_t*,...)	GrB_Scalar_extractElement_INT32(int32_t*,...)
GrB_Scalar_extractElement(uint32_t*,...)	GrB_Scalar_extractElement_UINT32(uint32_t*,...)
GrB_Scalar_extractElement(int64_t*,...)	GrB_Scalar_extractElement_INT64(int64_t*,...)
GrB_Scalar_extractElement(uint64_t*,...)	GrB_Scalar_extractElement_UINT64(uint64_t*,...)
GrB_Scalar_extractElement(float*,...)	GrB_Scalar_extractElement_FP32(float*,...)
GrB_Scalar_extractElement(double*,...)	GrB_Scalar_extractElement_FP64(double*,...)
GrB_Scalar_extractElement( <i>other</i> *,...)	GrB_Scalar_extractElement_UDT(void*,...)

Table 5.3: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Vector_build(...,const bool*,...)	GrB_Vector_build_BOOL(...,const bool*,...)
GrB_Vector_build(...,const int8_t*,...)	GrB_Vector_build_INT8(...,const int8_t*,...)
GrB_Vector_build(...,const uint8_t*,...)	GrB_Vector_build_UINT8(...,const uint8_t*,...)
GrB_Vector_build(...,const int16_t*,...)	GrB_Vector_build_INT16(...,const int16_t*,...)
GrB_Vector_build(...,const uint16_t*,...)	GrB_Vector_build_UINT16(...,const uint16_t*,...)
GrB_Vector_build(...,const int32_t*,...)	GrB_Vector_build_INT32(...,const int32_t*,...)
GrB_Vector_build(...,const uint32_t*,...)	GrB_Vector_build_UINT32(...,const uint32_t*,...)
GrB_Vector_build(...,const int64_t*,...)	GrB_Vector_build_INT64(...,const int64_t*,...)
GrB_Vector_build(...,const uint64_t*,...)	GrB_Vector_build_UINT64(...,const uint64_t*,...)
GrB_Vector_build(...,const float*,...)	GrB_Vector_build_FP32(...,const float*,...)
GrB_Vector_build(...,const double*,...)	GrB_Vector_build_FP64(...,const double*,...)
GrB_Vector_build(...,const <i>other</i> *,...)	GrB_Vector_build_UDT(...,const void*,...)
GrB_Vector_setElement(...,GrB_Scalar,...)	GrB_Vector_setElement_Scalar(...,const GrB_Scalar,...)
GrB_Vector_setElement(...,bool,...)	GrB_Vector_setElement_BOOL(..., bool,...)
GrB_Vector_setElement(...,int8_t,...)	GrB_Vector_setElement_INT8(..., int8_t,...)
GrB_Vector_setElement(...,uint8_t,...)	GrB_Vector_setElement_UINT8(..., uint8_t,...)
GrB_Vector_setElement(...,int16_t,...)	GrB_Vector_setElement_INT16(..., int16_t,...)
GrB_Vector_setElement(...,uint16_t,...)	GrB_Vector_setElement_UINT16(..., uint16_t,...)
GrB_Vector_setElement(...,int32_t,...)	GrB_Vector_setElement_INT32(..., int32_t,...)
GrB_Vector_setElement(...,uint32_t,...)	GrB_Vector_setElement_UINT32(..., uint32_t,...)
GrB_Vector_setElement(...,int64_t,...)	GrB_Vector_setElement_INT64(..., int64_t,...)
GrB_Vector_setElement(...,uint64_t,...)	GrB_Vector_setElement_UINT64(..., uint64_t,...)
GrB_Vector_setElement(...,float,...)	GrB_Vector_setElement_FP32(..., float,...)
GrB_Vector_setElement(...,double,...)	GrB_Vector_setElement_FP64(..., double,...)
GrB_Vector_setElement(..., <i>other</i> ,...)	GrB_Vector_setElement_UDT(...,const void*,...)
GrB_Vector_extractElement(GrB_Scalar,...)	GrB_Vector_extractElement_Scalar(GrB_Scalar,...)
GrB_Vector_extractElement(bool*,...)	GrB_Vector_extractElement_BOOL(bool*,...)
GrB_Vector_extractElement(int8_t*,...)	GrB_Vector_extractElement_INT8(int8_t*,...)
GrB_Vector_extractElement(uint8_t*,...)	GrB_Vector_extractElement_UINT8(uint8_t*,...)
GrB_Vector_extractElement(int16_t*,...)	GrB_Vector_extractElement_INT16(int16_t*,...)
GrB_Vector_extractElement(uint16_t*,...)	GrB_Vector_extractElement_UINT16(uint16_t*,...)
GrB_Vector_extractElement(int32_t*,...)	GrB_Vector_extractElement_INT32(int32_t*,...)
GrB_Vector_extractElement(uint32_t*,...)	GrB_Vector_extractElement_UINT32(uint32_t*,...)
GrB_Vector_extractElement(int64_t*,...)	GrB_Vector_extractElement_INT64(int64_t*,...)
GrB_Vector_extractElement(uint64_t*,...)	GrB_Vector_extractElement_UINT64(uint64_t*,...)
GrB_Vector_extractElement(float*,...)	GrB_Vector_extractElement_FP32(float*,...)
GrB_Vector_extractElement(double*,...)	GrB_Vector_extractElement_FP64(double*,...)
GrB_Vector_extractElement( <i>other</i> *,...)	GrB_Vector_extractElement_UDT(void*,...)
GrB_Vector_extractTuples(...,bool*,...)	GrB_Vector_extractTuples_BOOL(..., bool*,...)
GrB_Vector_extractTuples(...,int8_t*,...)	GrB_Vector_extractTuples_INT8(..., int8_t*,...)
GrB_Vector_extractTuples(...,uint8_t*,...)	GrB_Vector_extractTuples_UINT8(..., uint8_t*,...)
GrB_Vector_extractTuples(...,int16_t*,...)	GrB_Vector_extractTuples_INT16(..., int16_t*,...)
GrB_Vector_extractTuples(...,uint16_t*,...)	GrB_Vector_extractTuples_UINT16(..., uint16_t*,...)
GrB_Vector_extractTuples(...,int32_t*,...)	GrB_Vector_extractTuples_INT32(..., int32_t*,...)
GrB_Vector_extractTuples(...,uint32_t*,...)	GrB_Vector_extractTuples_UINT32(..., uint32_t*,...)
GrB_Vector_extractTuples(...,int64_t*,...)	GrB_Vector_extractTuples_INT64(..., int64_t*,...)
GrB_Vector_extractTuples(...,uint64_t*,...)	GrB_Vector_extractTuples_UINT64(..., uint64_t*,...)
GrB_Vector_extractTuples(...,float*,...)	GrB_Vector_extractTuples_FP32(..., float*,...)
GrB_Vector_extractTuples(...,double*,...)	GrB_Vector_extractTuples_FP64(..., double*,...)
GrB_Vector_extractTuples(..., <i>other</i> *,...)	GrB_Vector_extractTuples_UDT(..., void*,...)

Table 5.4: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Matrix_build(...,const bool*,...)	GrB_Matrix_build_BOOL(...,const bool*,...)
GrB_Matrix_build(...,const int8_t*,...)	GrB_Matrix_build_INT8(...,const int8_t*,...)
GrB_Matrix_build(...,const uint8_t*,...)	GrB_Matrix_build_UINT8(...,const uint8_t*,...)
GrB_Matrix_build(...,const int16_t*,...)	GrB_Matrix_build_INT16(...,const int16_t*,...)
GrB_Matrix_build(...,const uint16_t*,...)	GrB_Matrix_build_UINT16(...,const uint16_t*,...)
GrB_Matrix_build(...,const int32_t*,...)	GrB_Matrix_build_INT32(...,const int32_t*,...)
GrB_Matrix_build(...,const uint32_t*,...)	GrB_Matrix_build_UINT32(...,const uint32_t*,...)
GrB_Matrix_build(...,const int64_t*,...)	GrB_Matrix_build_INT64(...,const int64_t*,...)
GrB_Matrix_build(...,const uint64_t*,...)	GrB_Matrix_build_UINT64(...,const uint64_t*,...)
GrB_Matrix_build(...,const float*,...)	GrB_Matrix_build_FP32(...,const float*,...)
GrB_Matrix_build(...,const double*,...)	GrB_Matrix_build_FP64(...,const double*,...)
GrB_Matrix_build(...,const <i>other</i> *,...)	GrB_Matrix_build_UDT(...,const void*,...)
GrB_Matrix_setElement(...,GrB_Scalar,...)	GrB_Matrix_setElement_Scalar(...,const GrB_Scalar,...)
GrB_Matrix_setElement(...,bool,...)	GrB_Matrix_setElement_BOOL(..., bool,...)
GrB_Matrix_setElement(...,int8_t,...)	GrB_Matrix_setElement_INT8(..., int8_t,...)
GrB_Matrix_setElement(...,uint8_t,...)	GrB_Matrix_setElement_UINT8(..., uint8_t,...)
GrB_Matrix_setElement(...,int16_t,...)	GrB_Matrix_setElement_INT16(..., int16_t,...)
GrB_Matrix_setElement(...,uint16_t,...)	GrB_Matrix_setElement_UINT16(..., uint16_t,...)
GrB_Matrix_setElement(...,int32_t,...)	GrB_Matrix_setElement_INT32(..., int32_t,...)
GrB_Matrix_setElement(...,uint32_t,...)	GrB_Matrix_setElement_UINT32(..., uint32_t,...)
GrB_Matrix_setElement(...,int64_t,...)	GrB_Matrix_setElement_INT64(..., int64_t,...)
GrB_Matrix_setElement(...,uint64_t,...)	GrB_Matrix_setElement_UINT64(..., uint64_t,...)
GrB_Matrix_setElement(...,float,...)	GrB_Matrix_setElement_FP32(..., float,...)
GrB_Matrix_setElement(...,double,...)	GrB_Matrix_setElement_FP64(..., double,...)
GrB_Matrix_setElement(..., <i>other</i> ,...)	GrB_Matrix_setElement_UDT(...,const void*,...)
GrB_Matrix_extractElement(GrB_Scalar,...)	GrB_Matrix_extractElement_Scalar(GrB_Scalar,...)
GrB_Matrix_extractElement(bool*,...)	GrB_Matrix_extractElement_BOOL(bool*,...)
GrB_Matrix_extractElement(int8_t*,...)	GrB_Matrix_extractElement_INT8(int8_t*,...)
GrB_Matrix_extractElement(uint8_t*,...)	GrB_Matrix_extractElement_UINT8(uint8_t*,...)
GrB_Matrix_extractElement(int16_t*,...)	GrB_Matrix_extractElement_INT16(int16_t*,...)
GrB_Matrix_extractElement(uint16_t*,...)	GrB_Matrix_extractElement_UINT16(uint16_t*,...)
GrB_Matrix_extractElement(int32_t*,...)	GrB_Matrix_extractElement_INT32(int32_t*,...)
GrB_Matrix_extractElement(uint32_t*,...)	GrB_Matrix_extractElement_UINT32(uint32_t*,...)
GrB_Matrix_extractElement(int64_t*,...)	GrB_Matrix_extractElement_INT64(int64_t*,...)
GrB_Matrix_extractElement(uint64_t*,...)	GrB_Matrix_extractElement_UINT64(uint64_t*,...)
GrB_Matrix_extractElement(float*,...)	GrB_Matrix_extractElement_FP32(float*,...)
GrB_Matrix_extractElement(double*,...)	GrB_Matrix_extractElement_FP64(double*,...)
GrB_Matrix_extractElement( <i>other</i> ,...)	GrB_Matrix_extractElement_UDT(void*,...)
GrB_Matrix_extractTuples(..., bool*,...)	GrB_Matrix_extractTuples_BOOL(..., bool*,...)
GrB_Matrix_extractTuples(..., int8_t*,...)	GrB_Matrix_extractTuples_INT8(..., int8_t*,...)
GrB_Matrix_extractTuples(..., uint8_t*,...)	GrB_Matrix_extractTuples_UINT8(..., uint8_t*,...)
GrB_Matrix_extractTuples(..., int16_t*,...)	GrB_Matrix_extractTuples_INT16(..., int16_t*,...)
GrB_Matrix_extractTuples(..., uint16_t*,...)	GrB_Matrix_extractTuples_UINT16(..., uint16_t*,...)
GrB_Matrix_extractTuples(..., int32_t*,...)	GrB_Matrix_extractTuples_INT32(..., int32_t*,...)
GrB_Matrix_extractTuples(..., uint32_t*,...)	GrB_Matrix_extractTuples_UINT32(..., uint32_t*,...)
GrB_Matrix_extractTuples(..., int64_t*,...)	GrB_Matrix_extractTuples_INT64(..., int64_t*,...)
GrB_Matrix_extractTuples(..., uint64_t*,...)	GrB_Matrix_extractTuples_UINT64(..., uint64_t*,...)
GrB_Matrix_extractTuples(..., float*,...)	GrB_Matrix_extractTuples_FP32(..., float*,...)
GrB_Matrix_extractTuples(..., double*,...)	GrB_Matrix_extractTuples_FP64(..., double*,...)
GrB_Matrix_extractTuples(..., <i>other</i> *,...)	GrB_Matrix_extractTuples_UDT(..., void*,...)

Table 5.5: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Matrix_import(...,const bool*,...)	GrB_Matrix_import_BOOL(...,const bool*,...)
GrB_Matrix_import(...,const int8_t*,...)	GrB_Matrix_import_INT8(...,const int8_t*,...)
GrB_Matrix_import(...,const uint8_t*,...)	GrB_Matrix_import_UINT8(...,const uint8_t*,...)
GrB_Matrix_import(...,const int16_t*,...)	GrB_Matrix_import_INT16(...,const int16_t*,...)
GrB_Matrix_import(...,const uint16_t*,...)	GrB_Matrix_import_UINT16(...,const uint16_t*,...)
GrB_Matrix_import(...,const int32_t*,...)	GrB_Matrix_import_INT32(...,const int32_t*,...)
GrB_Matrix_import(...,const uint32_t*,...)	GrB_Matrix_import_UINT32(...,const uint32_t*,...)
GrB_Matrix_import(...,const int64_t*,...)	GrB_Matrix_import_INT64(...,const int64_t*,...)
GrB_Matrix_import(...,const uint64_t*,...)	GrB_Matrix_import_UINT64(...,const uint64_t*,...)
GrB_Matrix_import(...,const float*,...)	GrB_Matrix_import_FP32(...,const float*,...)
GrB_Matrix_import(...,const double*,...)	GrB_Matrix_import_FP64(...,const double*,...)
GrB_Matrix_import(...,const other,...)	GrB_Matrix_import_UDT(...,const void*,...)
GrB_Matrix_export(...,bool*,...)	GrB_Matrix_export_BOOL(...,bool*,...)
GrB_Matrix_export(...,int8_t*,...)	GrB_Matrix_export_INT8(...,int8_t*,...)
GrB_Matrix_export(...,uint8_t*,...)	GrB_Matrix_export_UINT8(...,uint8_t*,...)
GrB_Matrix_export(...,int16_t*,...)	GrB_Matrix_export_INT16(...,int16_t*,...)
GrB_Matrix_export(...,uint16_t*,...)	GrB_Matrix_export_UINT16(...,uint16_t*,...)
GrB_Matrix_export(...,int32_t*,...)	GrB_Matrix_export_INT32(...,int32_t*,...)
GrB_Matrix_export(...,uint32_t*,...)	GrB_Matrix_export_UINT32(...,uint32_t*,...)
GrB_Matrix_export(...,int64_t*,...)	GrB_Matrix_export_INT64(...,int64_t*,...)
GrB_Matrix_export(...,uint64_t*,...)	GrB_Matrix_export_UINT64(...,uint64_t*,...)
GrB_Matrix_export(...,float*,...)	GrB_Matrix_export_FP32(...,float*,...)
GrB_Matrix_export(...,double*,...)	GrB_Matrix_export_FP64(...,double*,...)
GrB_Matrix_export(...,other,...)	GrB_Matrix_export_UDT(...,void*,...)
GrB_free(GrB_Type*)	GrB_Type_free(GrB_Type*)
GrB_free(GrB_UnaryOp*)	GrB_UnaryOp_free(GrB_UnaryOp*)
GrB_free(GrB_IndexUnaryOp*)	GrB_IndexUnaryOp_free(GrB_IndexUnaryOp*)
GrB_free(GrB_BinaryOp*)	GrB_BinaryOp_free(GrB_BinaryOp*)
GrB_free(GrB_Monoid*)	GrB_Monoid_free(GrB_Monoid*)
GrB_free(GrB_Semiring*)	GrB_Semiring_free(GrB_Semiring*)
GrB_free(GrB_Scalar*)	GrB_Scalar_free(GrB_Scalar*)
GrB_free(GrB_Vector*)	GrB_Vector_free(GrB_Vector*)
GrB_free(GrB_Matrix*)	GrB_Matrix_free(GrB_Matrix*)
GrB_free(GrB_Descriptor*)	GrB_Descriptor_free(GrB_Descriptor*)
GrB_wait(GrB_Type, GrB_WaitMode)	GrB_Type_wait(GrB_Type, GrB_WaitMode)
GrB_wait(GrB_UnaryOp, GrB_WaitMode)	GrB_UnaryOp_wait(GrB_UnaryOp, GrB_WaitMode)
GrB_wait(GrB_IndexUnaryOp, GrB_WaitMode)	GrB_IndexUnaryOp_wait(GrB_IndexUnaryOp, GrB_WaitMode)
GrB_wait(GrB_BinaryOp, GrB_WaitMode)	GrB_BinaryOp_wait(GrB_BinaryOp, GrB_WaitMode)
GrB_wait(GrB_Monoid, GrB_WaitMode)	GrB_Monoid_wait(GrB_Monoid, GrB_WaitMode)
GrB_wait(GrB_Semiring, GrB_WaitMode)	GrB_Semiring_wait(GrB_Semiring, GrB_WaitMode)
GrB_wait(GrB_Scalar, GrB_WaitMode)	GrB_Scalar_wait(GrB_Scalar, GrB_WaitMode)
GrB_wait(GrB_Vector, GrB_WaitMode)	GrB_Vector_wait(GrB_Vector, GrB_WaitMode)
GrB_wait(GrB_Matrix, GrB_WaitMode)	GrB_Matrix_wait(GrB_Matrix, GrB_WaitMode)
GrB_wait(GrB_Descriptor, GrB_WaitMode)	GrB_Descriptor_wait(GrB_Descriptor, GrB_WaitMode)
GrB_error(const char**, const GrB_Type)	GrB_Type_error(const char**, const GrB_Type)
GrB_error(const char**, const GrB_UnaryOp)	GrB_UnaryOp_error(const char**, const GrB_UnaryOp)
GrB_error(const char**, const GrB_IndexUnaryOp)	GrB_IndexUnaryOp_error(const char**, const GrB_IndexUnaryOp)
GrB_error(const char**, const GrB_BinaryOp)	GrB_BinaryOp_error(const char**, const GrB_BinaryOp)
GrB_error(const char**, const GrB_Monoid)	GrB_Monoid_error(const char**, const GrB_Monoid)
GrB_error(const char**, const GrB_Semiring)	GrB_Semiring_error(const char**, const GrB_Semiring)
GrB_error(const char**, const GrB_Scalar)	GrB_Scalar_error(const char**, const GrB_Scalar)
GrB_error(const char**, const GrB_Vector)	GrB_Vector_error(const char**, const GrB_Vector)
GrB_error(const char**, const GrB_Matrix)	GrB_Matrix_error(const char**, const GrB_Matrix)
GrB_error(const char**, const GrB_Descriptor)	GrB_Descriptor_error(const char**, const GrB_Descriptor)

Table 5.6: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_eWiseMult(GrB_Vector,...,GrB_Semiring,...)	GrB_Vector_eWiseMult_Semiring(GrB_Vector,...,GrB_Semiring,...)
GrB_eWiseMult(GrB_Vector,...,GrB_Monoid,...)	GrB_Vector_eWiseMult_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_eWiseMult(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Vector_eWiseMult_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_eWiseMult(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_eWiseMult_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_eWiseMult(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_eWiseMult_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_eWiseMult(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_eWiseMult_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)
GrB_eWiseAdd(GrB_Vector,...,GrB_Semiring,...)	GrB_Vector_eWiseAdd_Semiring(GrB_Vector,...,GrB_Semiring,...)
GrB_eWiseAdd(GrB_Vector,...,GrB_Monoid,...)	GrB_Vector_eWiseAdd_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_eWiseAdd(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Vector_eWiseAdd_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_eWiseAdd(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_eWiseAdd_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_eWiseAdd(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_eWiseAdd_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_eWiseAdd(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_eWiseAdd_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)
GrB_extract(GrB_Vector,...,GrB_Vector,...)	GrB_Vector_extract(GrB_Vector,...,GrB_Vector,...)
GrB_extract(GrB_Matrix,...,GrB_Matrix,...)	GrB_Matrix_extract(GrB_Matrix,...,GrB_Matrix,...)
GrB_extract(GrB_Vector,...,GrB_Matrix,...)	GrB_Col_extract(GrB_Vector,...,GrB_Matrix,...)
GrB_assign(GrB_Vector,...,GrB_Vector,...)	GrB_Vector_assign(GrB_Vector,...,GrB_Vector,...)
GrB_assign(GrB_Matrix,...,GrB_Matrix,...)	GrB_Matrix_assign(GrB_Matrix,...,GrB_Matrix,...)
GrB_assign(GrB_Matrix,...,GrB_Vector,const GrB_Index*,...)	GrB_Col_assign(GrB_Matrix,...,GrB_Vector,const GrB_Index*,...)
GrB_assign(GrB_Matrix,...,GrB_Vector,GrB_Index,...)	GrB_Row_assign(GrB_Matrix,...,GrB_Vector,GrB_Index,...)
GrB_assign(GrB_Vector,...,GrB_Scalar,...)	GrB_Vector_assign_Scalar(GrB_Vector,...,const GrB_Scalar,...)
GrB_assign(GrB_Vector,...,bool,...)	GrB_Vector_assign_BOOL(GrB_Vector,..., bool,...)
GrB_assign(GrB_Vector,...,int8_t,...)	GrB_Vector_assign_INT8(GrB_Vector,..., int8_t,...)
GrB_assign(GrB_Vector,...,uint8_t,...)	GrB_Vector_assign_UINT8(GrB_Vector,..., uint8_t,...)
GrB_assign(GrB_Vector,...,int16_t,...)	GrB_Vector_assign_INT16(GrB_Vector,..., int16_t,...)
GrB_assign(GrB_Vector,...,uint16_t,...)	GrB_Vector_assign_UINT16(GrB_Vector,..., uint16_t,...)
GrB_assign(GrB_Vector,...,int32_t,...)	GrB_Vector_assign_INT32(GrB_Vector,..., int32_t,...)
GrB_assign(GrB_Vector,...,uint32_t,...)	GrB_Vector_assign_UINT32(GrB_Vector,..., uint32_t,...)
GrB_assign(GrB_Vector,...,int64_t,...)	GrB_Vector_assign_INT64(GrB_Vector,..., int64_t,...)
GrB_assign(GrB_Vector,...,uint64_t,...)	GrB_Vector_assign_UINT64(GrB_Vector,..., uint64_t,...)
GrB_assign(GrB_Vector,...,float,...)	GrB_Vector_assign_FP32(GrB_Vector,..., float,...)
GrB_assign(GrB_Vector,...,double,...)	GrB_Vector_assign_FP64(GrB_Vector,..., double,...)
GrB_assign(GrB_Vector,...,other,...)	GrB_Vector_assign_UDT(GrB_Vector,...,const void*,...)
GrB_assign(GrB_Matrix,...,GrB_Scalar,...)	GrB_Matrix_assign_Scalar(GrB_Matrix,...,const GrB_Scalar,...)
GrB_assign(GrB_Matrix,...,bool,...)	GrB_Matrix_assign_BOOL(GrB_Matrix,..., bool,...)
GrB_assign(GrB_Matrix,...,int8_t,...)	GrB_Matrix_assign_INT8(GrB_Matrix,..., int8_t,...)
GrB_assign(GrB_Matrix,...,uint8_t,...)	GrB_Matrix_assign_UINT8(GrB_Matrix,..., uint8_t,...)
GrB_assign(GrB_Matrix,...,int16_t,...)	GrB_Matrix_assign_INT16(GrB_Matrix,..., int16_t,...)
GrB_assign(GrB_Matrix,...,uint16_t,...)	GrB_Matrix_assign_UINT16(GrB_Matrix,..., uint16_t,...)
GrB_assign(GrB_Matrix,...,int32_t,...)	GrB_Matrix_assign_INT32(GrB_Matrix,..., int32_t,...)
GrB_assign(GrB_Matrix,...,uint32_t,...)	GrB_Matrix_assign_UINT32(GrB_Matrix,..., uint32_t,...)
GrB_assign(GrB_Matrix,...,int64_t,...)	GrB_Matrix_assign_INT64(GrB_Matrix,..., int64_t,...)
GrB_assign(GrB_Matrix,...,uint64_t,...)	GrB_Matrix_assign_UINT64(GrB_Matrix,..., uint64_t,...)
GrB_assign(GrB_Matrix,...,float,...)	GrB_Matrix_assign_FP32(GrB_Matrix,..., float,...)
GrB_assign(GrB_Matrix,...,double,...)	GrB_Matrix_assign_FP64(GrB_Matrix,..., double,...)
GrB_assign(GrB_Matrix,...,other,...)	GrB_Matrix_assign_UDT(GrB_Matrix,...,const void*,...)



Table 5.7: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_apply(GrB_Vector,...,GrB_UnaryOp,GrB_Vector,...)	GrB_Vector_apply(GrB_Vector,...,GrB_UnaryOp,GrB_Vector,...)
GrB_apply(GrB_Matrix,...,GrB_UnaryOp,GrB_Matrix,...)	GrB_Matrix_apply(GrB_Matrix,...,GrB_UnaryOp,GrB_Matrix,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Scalar,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_Scalar(GrB_Vector,...,GrB_BinaryOp,GrB_Scalar,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,bool,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_BOOL(GrB_Vector,...,GrB_BinaryOp,bool,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int8_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT8(GrB_Vector,...,GrB_BinaryOp,int8_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint8_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT8(GrB_Vector,...,GrB_BinaryOp,uint8_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int16_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT16(GrB_Vector,...,GrB_BinaryOp,int16_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint16_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT16(GrB_Vector,...,GrB_BinaryOp,uint16_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int32_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT32(GrB_Vector,...,GrB_BinaryOp,int32_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint32_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT32(GrB_Vector,...,GrB_BinaryOp,uint32_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int64_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT64(GrB_Vector,...,GrB_BinaryOp,int64_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint64_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT64(GrB_Vector,...,GrB_BinaryOp,uint64_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,float,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_FP32(GrB_Vector,...,GrB_BinaryOp,float,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,double,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_FP64(GrB_Vector,...,GrB_BinaryOp,double,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp, <i>other</i> ,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UDT(GrB_Vector,...,GrB_BinaryOp,const void*,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,GrB_Scalar,...)	GrB_Vector_apply_BinaryOp2nd_Scalar(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,GrB_Scalar,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,bool,...)	GrB_Vector_apply_BinaryOp2nd_BOOL(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,bool,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int8_t,...)	GrB_Vector_apply_BinaryOp2nd_INT8(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int8_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint8_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT8(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint8_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int16_t,...)	GrB_Vector_apply_BinaryOp2nd_INT16(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int16_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint16_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT16(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint16_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int32_t,...)	GrB_Vector_apply_BinaryOp2nd_INT32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int32_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint32_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint32_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int64_t,...)	GrB_Vector_apply_BinaryOp2nd_INT64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int64_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint64_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint64_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,float,...)	GrB_Vector_apply_BinaryOp2nd_FP32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,float,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,double,...)	GrB_Vector_apply_BinaryOp2nd_FP64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,double,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector, <i>other</i> ,...)	GrB_Vector_apply_BinaryOp2nd_UDT(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,const void*,...)

Table 5.8: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Scalar,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_Scalar(GrB_Matrix,...,GrB_BinaryOp,GrB_Scalar,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,bool,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_BOOL(GrB_Matrix,...,GrB_BinaryOp,bool,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int8_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT8(GrB_Matrix,...,GrB_BinaryOp,int8_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint8_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT8(GrB_Matrix,...,GrB_BinaryOp,uint8_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int16_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT16(GrB_Matrix,...,GrB_BinaryOp,int16_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint16_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT16(GrB_Matrix,...,GrB_BinaryOp,uint16_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int32_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT32(GrB_Matrix,...,GrB_BinaryOp,int32_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint32_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT32(GrB_Matrix,...,GrB_BinaryOp,uint32_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int64_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT64(GrB_Matrix,...,GrB_BinaryOp,int64_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint64_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT64(GrB_Matrix,...,GrB_BinaryOp,uint64_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,float,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_FP32(GrB_Matrix,...,GrB_BinaryOp,float,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,double,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_FP64(GrB_Matrix,...,GrB_BinaryOp,double,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp, <i>other</i> ,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UDT(GrB_Matrix,...,GrB_BinaryOp,const void*,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,GrB_Scalar,...)	GrB_Matrix_apply_BinaryOp2nd_Scalar(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,GrB_Scalar,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,bool,...)	GrB_Matrix_apply_BinaryOp2nd_BOOL(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,bool,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int8_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT8(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int8_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint8_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT8(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint8_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int16_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT16(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int16_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint16_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT16(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint16_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int32_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int32_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint32_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint32_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int64_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int64_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint64_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint64_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,float,...)	GrB_Matrix_apply_BinaryOp2nd_FP32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,float,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,double,...)	GrB_Matrix_apply_BinaryOp2nd_FP64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,double,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix, <i>other</i> ,...)	GrB_Matrix_apply_BinaryOp2nd_UDT(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,const void*,...)

Table 5.9: Long-name, nonpolymorphic form of GraphBLAS methods (continued).[\[Scott: NEW CONTENT\]](#)

Polymorphic signature	Nonpolymorphic signature
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)	GrB_Vector_apply_IndexOp_Scalar(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)	GrB_Vector_apply_IndexOp_BOOL(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)	GrB_Vector_apply_IndexOp_INT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)	GrB_Vector_apply_IndexOp_UINT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)	GrB_Vector_apply_IndexOp_INT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)	GrB_Vector_apply_IndexOp_UINT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)	GrB_Vector_apply_IndexOp_INT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)	GrB_Vector_apply_IndexOp_UINT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)	GrB_Vector_apply_IndexOp_INT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)	GrB_Vector_apply_IndexOp_UINT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)	GrB_Vector_apply_IndexOp_FP32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)	GrB_Vector_apply_IndexOp_FP64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector, <i>other</i> ,...)	GrB_Vector_apply_IndexOp_UDT(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,const void*,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)	GrB_Matrix_apply_IndexOp_Scalar(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)	GrB_Matrix_apply_IndexOp_BOOL(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)	GrB_Matrix_apply_IndexOp_INT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)	GrB_Matrix_apply_IndexOp_UINT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)	GrB_Matrix_apply_IndexOp_INT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)	GrB_Matrix_apply_IndexOp_UINT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)	GrB_Matrix_apply_IndexOp_INT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)	GrB_Matrix_apply_IndexOp_UINT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)	GrB_Matrix_apply_IndexOp_INT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)	GrB_Matrix_apply_IndexOp_UINT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)	GrB_Matrix_apply_IndexOp_FP32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)	GrB_Matrix_apply_IndexOp_FP64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix, <i>other</i> ,...)	GrB_Matrix_apply_IndexOp_UDT(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,const void*,...)

Table 5.10: Long-name, nonpolymorphic form of GraphBLAS methods (continued).[\[Scott: NEW CONTENT\]](#)

Polymorphic signature	Nonpolymorphic signature
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)</code>	<code>GrB_Vector_select_Scalar(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)</code>	<code>GrB_Vector_select_BOOL(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)</code>	<code>GrB_Vector_select_INT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)</code>	<code>GrB_Vector_select_UINT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)</code>	<code>GrB_Vector_select_INT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)</code>	<code>GrB_Vector_select_UINT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)</code>	<code>GrB_Vector_select_INT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)</code>	<code>GrB_Vector_select_UINT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)</code>	<code>GrB_Vector_select_INT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)</code>	<code>GrB_Vector_select_UINT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)</code>	<code>GrB_Vector_select_FP32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)</code>	<code>GrB_Vector_select_FP64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,other,...)</code>	<code>GrB_Vector_select_UDT(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,const void*,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)</code>	<code>GrB_Matrix_select_Scalar(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)</code>	<code>GrB_Matrix_select_BOOL(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)</code>	<code>GrB_Matrix_select_INT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)</code>	<code>GrB_Matrix_select_UINT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)</code>	<code>GrB_Matrix_select_INT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)</code>	<code>GrB_Matrix_select_UINT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)</code>	<code>GrB_Matrix_select_INT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)</code>	<code>GrB_Matrix_select_UINT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)</code>	<code>GrB_Matrix_select_INT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)</code>	<code>GrB_Matrix_select_UINT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)</code>	<code>GrB_Matrix_select_FP32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)</code>	<code>GrB_Matrix_select_FP64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,other,...)</code>	<code>GrB_Matrix_select_UDT(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,const void*,...)</code>

Table 5.11: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_reduce(GrB_Vector,...,GrB_Monoid,...)	GrB_Matrix_reduce_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_reduce(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Matrix_reduce_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_reduce(GrB_Scalar,...,GrB_Monoid,GrB_Vector,...)	GrB_Vector_reduce_Monoid_Scalar(GrB_Scalar,...,GrB_Vector,...)
GrB_reduce(GrB_Scalar,...,GrB_BinaryOp,GrB_Vector,...)	GrB_Vector_reduce_BinaryOp_Scalar(GrB_Scalar,...,GrB_Vector,...)
GrB_reduce(bool*,...,GrB_Vector,...)	GrB_Vector_reduce_BOOL(bool*,...,GrB_Vector,...)
GrB_reduce(int8_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT8(int8_t*,...,GrB_Vector,...)
GrB_reduce(uint8_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT8(uint8_t*,...,GrB_Vector,...)
GrB_reduce(int16_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT16(int16_t*,...,GrB_Vector,...)
GrB_reduce(uint16_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT16(uint16_t*,...,GrB_Vector,...)
GrB_reduce(int32_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT32(int32_t*,...,GrB_Vector,...)
GrB_reduce(uint32_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT32(uint32_t*,...,GrB_Vector,...)
GrB_reduce(int64_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT64(int64_t*,...,GrB_Vector,...)
GrB_reduce(uint64_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT64(uint64_t*,...,GrB_Vector,...)
GrB_reduce(float*,...,GrB_Vector,...)	GrB_Vector_reduce_FP32(float*,...,GrB_Vector,...)
GrB_reduce(double*,...,GrB_Vector,...)	GrB_Vector_reduce_FP64(double*,...,GrB_Vector,...)
GrB_reduce( <i>other</i> *,...,GrB_Vector,...)	GrB_Vector_reduce_UDT(void*,...,GrB_Vector,...)
GrB_reduce(GrB_Scalar,...,GrB_Monoid,GrB_Matrix,...)	GrB_Matrix_reduce_Monoid_Scalar(GrB_Scalar,...,GrB_Monoid,GrB_Matrix,...)
GrB_reduce(GrB_Scalar,...,GrB_BinaryOp,GrB_Matrix,...)	GrB_Matrix_reduce_BinaryOp_Scalar(GrB_Scalar,...,GrB_BinaryOp,GrB_Matrix,...)
GrB_reduce(bool*,...,GrB_Matrix,...)	GrB_Matrix_reduce_BOOL(bool*,...,GrB_Matrix,...)
GrB_reduce(int8_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT8(int8_t*,...,GrB_Matrix,...)
GrB_reduce(uint8_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT8(uint8_t*,...,GrB_Matrix,...)
GrB_reduce(int16_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT16(int16_t*,...,GrB_Matrix,...)
GrB_reduce(uint16_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT16(uint16_t*,...,GrB_Matrix,...)
GrB_reduce(int32_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT32(int32_t*,...,GrB_Matrix,...)
GrB_reduce(uint32_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT32(uint32_t*,...,GrB_Matrix,...)
GrB_reduce(int64_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT64(int64_t*,...,GrB_Matrix,...)
GrB_reduce(uint64_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT64(uint64_t*,...,GrB_Matrix,...)
GrB_reduce(float*,...,GrB_Matrix,...)	GrB_Matrix_reduce_FP32(float*,...,GrB_Matrix,...)
GrB_reduce(double*,...,GrB_Matrix,...)	GrB_Matrix_reduce_FP64(double*,...,GrB_Matrix,...)
GrB_reduce( <i>other</i> *,...,GrB_Matrix,...)	GrB_Matrix_reduce_UDT(void*,...,GrB_Matrix,...)
GrB_kronecker(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_kronecker_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_kronecker(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_kronecker_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_kronecker(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_kronecker_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)



# 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.

- 7491 • (Issues BB-25, GH-42) Added import and export methods for matrices to/from API specified  
7492 formats. Three formats have been specified: CSC, CSR, COO. Dense row and column formats  
7493 have been deferred.
- 7494 • (Issue BB-75) Added matrix constructor to build a diagonal `GrB_Matrix` from a `GrB_Vector`.
- 7495 • (Issue BB-73) Allow `GrB_NULL` for dup operator in matrix and vector `build` methods. Return  
7496 error if duplicate locations encountered.
- 7497 • (Issue BB-58) Added matrix and vector methods to remove (annihilate) elements.
- 7498 • (Issue BB-17) Added `GrB_ABS_T` (absolute value) unary operator.
- 7499 • (Issue GH-46) Adding `GrB_ONEB_T` binary operator that returns 1 cast to type T (not to  
7500 be confused with the proposed unary operator).
- 7501 • (Issue GH-53) Added language about what constitutes a “conformant” implementation. Added  
7502 `GrB_NOT_IMPLEMENTED` return value (API error) for API any combinations of inputs to  
7503 a method that is not supported by the implementation.
- 7504 • Added `GrB_EMPTY_OBJECT` return value (execution error) that is used when an opaque  
7505 object (currently only `GrB_Scalar`) is passed as an input that cannot be empty.
- 7506 • (Issue BB-45) Removed language about annihilators.
- 7507 • (Issue BB-69) Made names/symbols containing underscores searchable in PDF.
- 7508 • Updated a number algorithms in the appendix to use new operations and methods.
- 7509 • Numerous additions (some changes) to the non-polymorphic interface to track changes to the  
7510 specification.
- 7511 • Typographical error in version macros was corrected. They are all caps: `GRB_VERSION` and  
7512 `GRB_SUBVERSION`.
- 7513 • Typographical change to `eWiseAdd` Description to be consistent in order of set intersections.
- 7514 • Typographical errors in `eWiseAdd`: cut-and-paste errors from `eWiseMult`/set intersection  
7515 fixed to read `eWiseAdd`/set union.
- 7516 • Typographical error (`NEQ`  $\rightarrow$  `NE`) in Description of Table 3.8.

7517 Changes in 1.3.0 (Released: 25 September 2019):

- 7518 • (Issue BB-50) Changed definition of completion and added `GrB_wait()` that takes an opaque  
7519 GraphBLAS object as an argument.
- 7520 • (Issue BB-39) Added `GrB_kronecker` operation.
- 7521 • (Issue BB-40) Added variants of the `GrB_apply` operation that take a binary function and a  
7522 scalar.



7523  
7524  
  
7525  
  
7526  
7527  
  
7528  
7529  
  
7530  
  
7531  
7532  
  
7533  
7534  
  
7535  
7536  
  
7537  
  
7538  
7539  
  
7540  
  
7541  
7542  
  
7543  
7544  
  
7545  
  
7546  
7547  
  
7548  
  
7549  
7550  
  
7551  
7552  
  
7553  
  
7554  
  
7555

- (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.

7556 • Fixed miscellaneous typographical errors (such as  $\otimes$ ,  $\oplus$ ).

7557 Changes in 1.2.0:

7558 • Removed "provisional" clause.

7559 Changes in 1.1.0:

7560 • Removed unnecessary `const` from `nindices`, `nrows`, and `ncols` parameters of both `extract` and  
7561 `assign` operations.

7562 • Signature of `GrB_UnaryOp_new` changed: order of input parameters changed.

7563 • Signature of `GrB_BinaryOp_new` changed: order of input parameters changed.

7564 • Signature of `GrB_Monoid_new` changed: removal of domain argument which is now inferred  
7565 from the domains of the binary operator provided.

7566 • Signature of `GrB_Vector_extractTuples` and `GrB_Matrix_extractTuples` to add an in/out ar-  
7567 gument, `n`, which indicates the size of the output arrays provided (in terms of number of  
7568 elements, not number of bytes). Added new execution error, `GrB_INSUFFICIENT_SPACE`  
7569 which is returned when the capacities of the output arrays are insufficient to hold all of the  
7570 tuples.

7571 • Changed `GrB_Column_assign` to `GrB_Col_assign` for consistency in non-polymorphic inter-  
7572 face.

7573 • Added replace flag (`z`) notation to Table 4.1.

7574 • Updated the "Mathematical Description" of the `assign` operation in Table 4.1.

7575 • Added triangle counting example.

7576 • Added subsection headers for `accumulate` and `mask/replace` discussions in the Description  
7577 sections of GraphBLAS operations when the respective text was the "standard" text (i.e.,  
7578 identical in a majority of the operations).

7579 • Fixed typographical errors.

7580 Changes in 1.0.2:

7581 • Expanded the definitions of `Vector_build` and `Matrix_build` to conceptually use intermediate  
7582 matrices and avoid casting issues in certain implementations.

7583 • Fixed the bug in the `GrB_assign` definition. Elements of the output object are no longer being  
7584 erased outside the assigned area.

7585 • Changes non-polymorphic interface:

7586 – Renamed `GrB_Row_extract` to `GrB_Col_extract`.

- 7587           – Renamed GrB\_Vector\_reduce\_BinaryOp to GrB\_Matrix\_reduce\_BinaryOp.
- 7588           – Renamed GrB\_Vector\_reduce\_Monoid to GrB\_Matrix\_reduce\_Monoid.
- 7589       • Fixed the bugs with respect to isolated vertices in the Maximal Independent Set example.
- 7590       • 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.

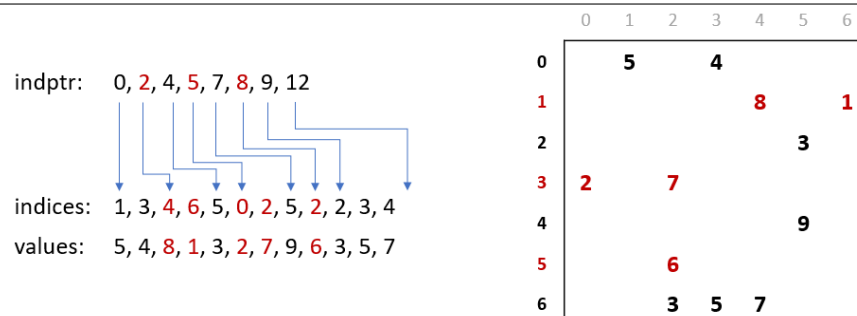


Figure B.1: Data layout for CSR format.

### 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.

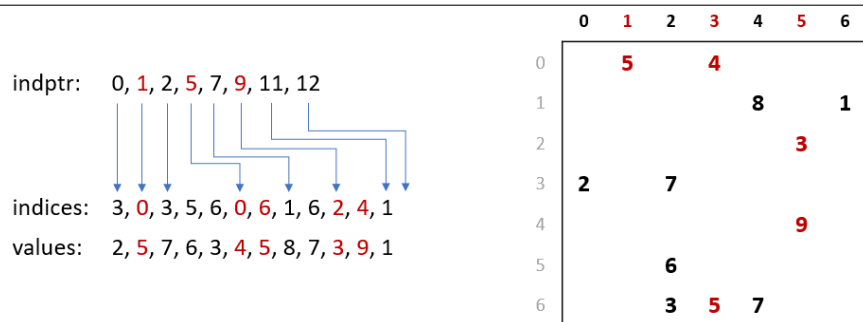


Figure B.2: Data layout for CSC format.

### 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.

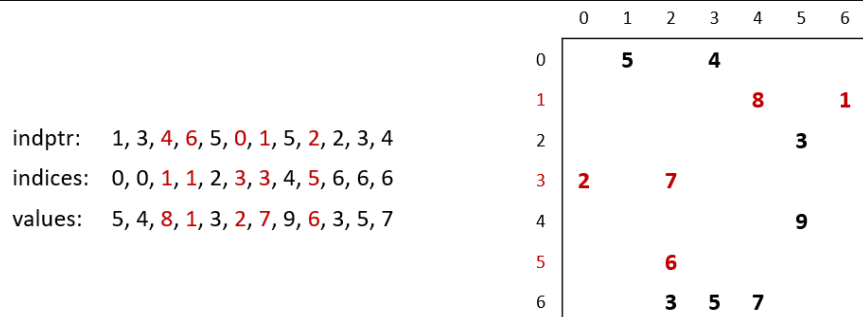


Figure B.3: Data layout for COO format.

## 7623 Appendix C

## 7624 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  $c = c \ominus !new$ 
75 GrB_eWiseMult(candidates, new_members, GrB_NULL,
76               GrB_LAND, candidates, candidates, GrB_DESC_RC);
77
78 GrB_Vector_nvals(&nvals, candidates);
79 if (nvals == 0) { break; } // early exit condition
80
81 // Neighbors of new members can also be removed from candidates
82 GrB_mnv(new_neighbors, candidates, GrB_NULL, GrB_LOR_LAND_SEMIRING_BOOL,
83         A, new_members, GrB_NULL);
84 GrB_eWiseMult(candidates, new_neighbors, GrB_NULL, GrB_LAND,
85               candidates, candidates, GrB_DESC_RC);
86
87 GrB_Vector_nvals(&nvals, candidates);
88 }
89
90 GrB_free(&neighbor_max); // free all objects "new'ed"
91 GrB_free(&new_members);
92 GrB_free(&new_neighbors);
93 GrB_free(&prob);
94 GrB_free(&candidates);
95 GrB_free(&set_random);
96 GrB_free(&degrees);
97
98 return GrB_SUCCESS;
99 }

```



## C.7 Example: Counting triangles in GraphBLAS

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <stdint.h>
4 #include <stdbool.h>
5 #include "GraphBLAS.h"
6
7 /*
8  * Given an  $n \times n$  boolean adjacency matrix,  $A$ , of an undirected graph, computes
9  * the number of triangles in the graph.
10 */
11 uint64_t triangle_count(GrB_Matrix A)
12 {
13     GrB_Index n;
14     GrB_Matrix_nrows(&n, A);           //  $n = \#$  of vertices
15
16     //  $L$ :  $N \times N$ , lower-triangular, bool
17     GrB_Matrix L;
18     GrB_Matrix_new(&L, GrB_BOOL, n, n);
19     GrB_select(L, GrB_NULL, GrB_NULL, GrB_TRIL, A, 0UL, GrB_NULL);
20
21     GrB_Matrix C;
22     GrB_Matrix_new(&C, GrB_UINT64, n, n);
23
24     GrB_mxm(C, L, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_UINT64, L, L, GrB_NULL); //  $C \langle L \rangle = L +.* L$ 
25
26     uint64_t count;
27     GrB_reduce(&count, GrB_NULL, GrB_PLUS_MONOID_UINT64, C, GrB_NULL); // 1-norm of  $C$ 
28
29     GrB_free(&C);
30     GrB_free(&L);
31
32     return count;
33 }
```