Proposal for a GraphBLAS C API

(*Working document from the* GraphBLAS *Signatures Subgroup*)

Aydin Buluc, Timothy Mattson, Scott McMillan, José Moreira, Carl Yang

GraphBLAS Signatures Subgroup

Generated on 2016/02/16 at 15:00:19 EDT

[Carl: testing] [Scott: testing] [Aydin: testing] [Tim: testing] [Jose: testing] [Yzelman: testing]

1 Introduction

This is a proposal for the C programming language binding of the GraphBLAS interface. We adopt C99 as the standard definition of the C programming language. Furthermore, we assume that the language has been extended with the `_Generic` construct from C11. After establishing some basic concepts, we proceed by describing the objects in GraphBLAS: spaces, vectors, matrices and descriptors. We then describe the various methods that operate on those objects. The appendix includes examples of GraphBLAS in C.

[Yzelman: Is the use of _Generic an implementation detail or will it show up in the API? If the former, could it be dropped from the spec?]

2 Basic concepts

2.1 Domains

GraphBLAS defines two kinds of collections: matrices and vectors. For any given collection, the elements of the collection belong to a *domain*, which is the set of valid values for the element. In GraphBLAS, domains correspond to the valid values for types from the host language (in our case, the C programming language). For any variable or object V in GraphBLAS we denote as $D(V)$ the domain of V. That is, the set of possible values that elements of V can take. The predefined types, and corresponding domains, of GraphBLAS are shown in Table 1. The Boolean type is defined in `stdbool.h`, the integral types are defined in `stdint.h`, and the floating-point types are native to the language. GraphBLAS also supports user defined types. In that case, the domain is the set of valid values for a variable of that type.

Table 1: Predefined types and corresponding domains for GraphBLAS in C. [Scott: Domains and types are not always equivalent although they may be in this table and the third column of this table does not indicate that these items (are they enums?) really are indicating domains. This is a naming nitpick, sorry. I think we should also go through the exercise of supporting a Domain that does not have a 1-to-1 correllation with any type e.g. {0,1}. Also need to understand how a user extends the set of domains supported.] [Aydin: There will be a way to introduce new GraphBLAS identifiers, similar to MPI_Type_Commit, these are just predefined stuff]

type	domain	GraphBLAS identifier
`bool`	{`false`, `true`}	GrB_BOOL
`int8_t`	$\mathbb{Z} \cap [-2^7, 2^7)$	GrB_INT8
`uint8_t`	$\mathbb{Z} \cap [0, 2^8)$	GrB_UINT8
`int16_t`	$\mathbb{Z} \cap [-2^{15}, 2^{15})$	GrB_INT16
`uint16_t`	$\mathbb{Z} \cap [0, 2^{16})$	GrB_UINT16
`int32_t`	$\mathbb{Z} \cap [-2^{31}, 2^{31})$	GrB_INT32
`uint32_t`	$\mathbb{Z} \cap [0, 2^{32})$	GrB_UINT32
`int64_t`	$\mathbb{Z} \cap [-2^{63}, 2^{63})$	GrB_INT64
`uint64_t`	$\mathbb{Z} \cap [0, 2^{64})$	GrB_UINT64
`float`	IEEE 754 binary32	GrB_FLOAT
`double`	IEEE 754 binary64	GrB_DOUBLE

2.2 Operations

In GraphBLAS, a *binary operation* is a function that maps two input values to one output value. A *unary operation* is a function that maps one input value to one output value. The value of the output is uniquely determined by the value of the input(s). Binary functions are defined over two input domains and produce an output from a (possibly different) third domain. Unary functions are specified over one input domain and produce an output from a (possibly different) second domain. The predefined operations of GraphBLAS are listed in Table 2.

3 Objects

3.1 Spaces

[Scott: Space is too overloaded. We should decide on a better name. I propose calling the section "Algebraic Structures" or"Algebra" for short.]

A GraphBLAS *space* $S = \langle D_1, D_2, D_3, \oplus, \otimes, \mathbf{0}[, \mathbf{1}]\rangle$ is defined by three domains D_1, D_2 and D_3, an additive operation $\oplus : D_3 \times D_3 \to D_3$, with corresponding identity $\mathbf{0} \in D_3$, and a multiplicative operation $\otimes : D_1 \times D_2 \to D_3$, with optional corresponding identity $\mathbf{1} \in D_1 \cap D_2$. If $\mathbf{1}$ is specified, then $D_1 \subseteq D_3$ and $D_2 \subseteq D_3$ must be satisfied. For a given GraphBLAS space $S = \langle D_1, D_2, D_3, \oplus, \otimes, \mathbf{0}, \mathbf{1}\rangle$ we define $D_1(S) = D_1$, $D_2(S) = D_2$, $D_3(S) = D_3$, $\oplus(S) = \oplus$, $\otimes(S) = \otimes$, $\mathbf{0}(S) = \mathbf{0}$ and $\mathbf{1}(S) = \mathbf{1}$. We note that, in the special case of $D_1 = D_2 = D_3$ and $\mathbf{1}$ defined, a GraphBLAS space reduces to

Table 2: Predefined operations for GraphBLAS in C. (Just a sample.) [Scott: Rather than a sample we should strive to specify the complete set of predefined operations; add min and max. What C construct are the symbols in the second column. They look like enums or constants that need to be completely specified by the API.]

kind	operation	domains	description
	GrB_NOP		no operation
unary	GrB_LNOT	`bool` → `bool`	logical inverse
binary	GrB_LAND	`bool` × `bool` → `bool`	logical AND
binary	GrB_LOR	`bool` × `bool` → `bool`	logical OR
binary	GrB_LXOR	`bool` × `bool` → `bool`	logical XOR
binary	GrB_PLUS	`int64_t` × `int64_t` → `int64_t`	signed integer addition
binary	GrB_PLUS	`uint64_t` × `uint64_t` → `uint64_t`	unsigned integer addition
binary	GrB_PLUS	`double` × `double` → `double`	floating-point addition
binary	GrB_MINUS	`int64_t` × `int64_t` → `int64_t`	signed integer subtraction
binary	GrB_MINUS	`uint64_t` × `uint64_t` → `uint64_t`	unsigned integer subtraction
binary	GrB_MINUS	`double` × `double` → `double`	floating-point subtraction
binary	GrB_TIMES	`int64_t` × `int64_t` → `int64_t`	signed integer multiplication
binary	GrB_TIMES	`uint64_t` × `uint64_t` → `uint64_t`	unsigned integer multiplication
binary	GrB_TIMES	`double` × `double` → `double`	floating-point multiplication
binary	GrB_DIV	`int64_t` × `int64_t` → `int64_t`	signed integer division
binary	GrB_DIV	`uint64_t` × `uint64_t` → `uint64_t`	unsigned integer division
binary	GrB_DIV	`double` × `double` → `double`	floating-point division
binary	GrB_EQ	`int64_t` × `int64_t` → `bool`	signed integer equal
binary	GrB_EQ	`uint64_t` × `uint64_t` → `bool`	unsigned integer equal
binary	GrB_EQ	`double` × `double` → `bool`	floating-point equal
binary	GrB_NE	`int64_t` × `int64_t` → `bool`	signed integer not equal
binary	GrB_NE	`uint64_t` × `uint64_t` → `bool`	unsigned integer not equal
binary	GrB_NE	`double` × `double` → `bool`	floating-point not equal
binary	GrB_GT	`int64_t` × `int64_t` → `bool`	signed integer greater than
binary	GrB_GT	`uint64_t` × `uint64_t` → `bool`	unsigned integer greater than
binary	GrB_GT	`double` × `double` → `bool`	floating-point greater than
binary	GrB_LT	`int64_t` × `int64_t` → `bool`	signed integer less than
binary	GrB_LT	`uint64_t` × `uint64_t` → `bool`	unsigned integer less than
binary	GrB_LT	`double` × `double` → `bool`	floating-point less than
binary	GrB_GE	`int64_t` × `int64_t` → `bool`	signed integer greater than or equal
binary	GrB_GE	`uint64_t` × `uint64_t` → `bool`	unsigned integer greater than or equal
binary	GrB_GE	`double` × `double` → `bool`	floating-point greater than or equal
binary	GrB_LE	`int64_t` × `int64_t` → `bool`	signed integer less than or equal
binary	GrB_LE	`uint64_t` × `uint64_t` → `bool`	unsigned integer less than or equal
binary	GrB_LE	`double` × `double` → `bool`	floating-point less than or equal

the conventional *semiring* algebraic structure [Scott: and **0** must be $\oplus$ identity and $\otimes$ annihilator in order to be a semiring as well].

[Yzelman: D_i, etc is overloaded or used recursively]

[Yzelman: There could be 2 **1**'s: one in each of D_1 and D_2 and both could be optional. This would relax the constraint that $D_1 \subseteq D_3$ and $D_2 \subseteq D_3$.]

[Yzelman: more far-fetched: a Space could consist of four domains: $\otimes : D_1 \times D_2 \to D_3$, and $\oplus : D_3 \times D_3 \to D_4$ where D_3 can be reduced to something in D_4.]

[Scott: I think we should consider another object which is a monoid: a binary function and its identity. This space or algebraic structure could then be built from two monoids. Further the monoids have three domains each: two for input and one for output. There are some things to discuss: This implies a more relaxation of the "space" to up to six domains if implicit or explicit casting allows it.]

[Scott: I think the Monoid is what is most natural to provide to ewiseadd and ewisemult.]

3.2 Vectors

A vector $\mathbf{v} = \langle D, N, \{(i, \mathbf{v}(i))\}\rangle$ is defined by a domain D, a size $N > 0$ and a set of tuples $(i, \mathbf{v}(i))$ where $0 \leq i < N$ and $\mathbf{v}(i) \in D$. A particular value of i can only appear at most once in $\mathbf{v}$. We define $n(\mathbf{v}) = N$ and $L(\mathbf{v}) = \{(i, \mathbf{v}(i))\}$. We also define the set $\mathbf{i}(\mathbf{v}) = \{i : (i, \mathbf{v}(i)) \in \mathbf{v}\}$, and $D(\mathbf{v}) = D$.

[Yzelman: Overloaded use of $\mathbf{v}$ in the definition.]

[Yzelman: Why not define in terms of D^N]

[Scott: Going out on a limb here: matrices and vectors should be thought of more as storage containers rather than the pure mathematical concepts. In this context I need a compelling reason to have vectors *and* matrices. The only good reason I can think to to have vectors is that they are strictly one dimensional; i.e., saves storage not having to store a second index for each stored value. Implication: if you want to perform multiple BFS traversals, it should be with a Matrix of wavefronts and not a 2D stack of 'vectors'. Further if we have a strictly 1D structure, I believe we must give it an implicit orientation for matrix operations (I prefer column vector) and discuss all the other vector operations proposed a while back.]

3.3 Matrices

A matrix $\mathbf{A} = \langle D, M, N, \{(i, j, \mathbf{A}(i, j))\}\rangle$ is defined by a domain D. its number of rows $M > 0$, its number of columns $N > 0$ and a set of tuples $(i, j, \mathbf{A}(i, j))$ where $0 \leq i < M$, $0 \leq j < N$, and $\mathbf{A}(i, j)i \in D$. A particular pair of values i, j can only appear at most once in $\mathbf{A}$. We define $n(\mathbf{A}) = N$, $m(\mathbf{A}) = M$ and $L(\mathbf{A}) = \{(i, j, \mathbf{A}(i, j))\}$. We also define the sets $\mathbf{i}(\mathbf{A}) = \{i : \exists(i, j, \mathbf{A}(i, j)) \in \mathbf{A}\}$ and $\mathbf{j}(\mathbf{A}) = \{j : \exists(i, j, \mathbf{A}(i, j)) \in \mathbf{A}\}$. (These are the sets of nonempty rows and columns of $\mathbf{A}$, respectively.) Finally, $D(\mathbf{A}) = D$.

[Yzelman: Overloaded use of $\mathbf{A}$ in the definition.]

[Yzelman: Why not define in terms of $D^{M \times N}$.]

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

3.4 Descriptors

Descriptors are used as input parameters in various GraphBLAS methods to provide more details of the operation to be performed by those methods. In particular, descriptors specify how the other input parameters should be processed before the main operation of a method is performed. For example, a descriptor may specify that a particular input matrix needs to be transposed or that a mask needs to be inverted before using it in the operation. Some methods may also allow additional processing of the result before generating the final output parameter.

[Scott: 'inverted' above is ambiguous, we need to define a better term like "structural complement". Further we should specify behaviour if the mask is a dense container not just when it is sparse.]

For the purpose of constructing descriptors, the parameters of a method are identified by specific names. The output parameter (typically the first parameter in a GraphBLAS method) is OUTP. The input parameters are named ARG0, ARG1, ARG2 and so on from the first input parameter to the last. The mask (typically the next to last parameter in a method) is named MASK. Finally, the descriptor (typically the last parameter in a method) is not named, since GraphBLAS does not support modifications of descriptors themselves.

[Scott: Is negate/invert/structural complement only restricted to masks?]

[Scott: We must specify the behaviour of the descriptor's transpose. E.g. is it allowed to mutate the operand for the duration of the operation, or is this strictly a flag that affects the operation only – how it accesses the operand's values?]

4 Methods

4.1 Vector-matrix multiply (vxm)

Multiplies a vector by a matrix within a space. The result is a vector.

C99 Syntax

```
#include "GraphBLAS.h"
GrB_info GrB_vxm(GrB_Vector *u, const GrB_Space s, const GrB_Vector v,
                 const GrB_Matrix A, const GrB_Vector m, const GrB_Descriptor d)
```

[Yzelman: Should `u` parameter have `restrict` keyword?]

Input Parameters

- s (ARG0) Space used in the vector-matrix multiply.
- v (ARG1) Vector to be multiplied.
- A (ARG2) Matrix to be multiplied.
- m (MASK) Operation mask. The mask specifies which elements of the result vector are to be computed. If no mask is necessary (i.e., compute all elements of result vector), GrB_NULL can be used.
- d Operation descriptor. The descriptor is used to specify details of the operation, such as transpose the matrix or not, invert the mask or not (see below). If a *default* descriptor is desired, GrB_NULL can be used.

[Yzelman: Can v and m refer to the same container?] [Yzelman: Should (u and v) or (u and m NOT refer to the same container...use restrict?]

Output Parameter

- u (OUTP) Address of result vector.

Return Value

[Scott: We should specify anything that we can about the behaviour/program state when any error condition is returned. What gaurantees are we giving (think consistency models)? Definitely related to mutability question earlier.]

GrB_SUCCESS	operation completed successfully
GrB_PANIC	unknown internal error
GrB_OUTOFMEM	not enough memory available for operation
GrB_MISMATCH	mismatch among vectors, matrix and/or space

[Scott: More return values possible: domain/type mismatches, dimension mismatches]

Description

Vectors $\mathbf{v}, \mathbf{m}$ and matrix $\mathbf{A}$ are computed from input parameters v, m and A, respectively, as specified by descriptor d. (See below for the properties of a descriptor. In the simplest form, these are just copies, but additional preprocessing, including casting, can be specified.) $D(\mathbf{v}) \equiv D_1(\mathsf{s})$ and $D(\mathbf{A}) \equiv D_2(\mathsf{s})$. If m is GrB_NULL then $\mathbf{m}$ is a Boolean vector of size $n(\mathbf{A})$ and with all elements set to true.

A consistency check is performed to verify that $n(\mathbf{v}) = m(\mathbf{A})$ and $n(\mathbf{m}) = n(\mathbf{A})$. If a consistency check fails, the operation is aborted and the method returns GrB_MISMATCH.

A new vector $\mathbf{u} = \langle D_3(\mathsf{s}), n(\mathbf{A}), L(\mathbf{u}) = \{(i, \mathbf{u}(i)) : \mathbf{m}(i) = \mathsf{true}\}\rangle$ is created. The value of each of its elements is computed by $\mathbf{u}(i) = \bigoplus_{j \in \mathbf{i}(\mathbf{v}) \cap \mathbf{i}(\mathbf{A}(:,i))} (\mathbf{v}(j) \otimes \mathbf{A}(j,i))$, where $\oplus$ and $\otimes$ are the additive and multiplicative operations of space s, respectively. If $\mathbf{i}(\mathbf{v}) \cap \mathbf{i}(\mathbf{A}(:,i)) = \emptyset$ then the pair $(i, \mathbf{u}(i))$ is not included in $L(\mathbf{u})$.

Finally, output parameter u is computed from vector **u** as specified by descriptor d. (Again, in the simplest case this is just a copy, but additional postprocessing, including casting and accumulation of result values, can be specified.) A consistency check is performed to verify that $n(\mathsf{u}) = n(\mathbf{u})$. If the consistency check fails, the operation is aborted and the method return GrB_MISMATCH.

[Scott: We need a more explicit discussion/specification regarding masks and accumulation and their interaction (perhaps the diagram Manoj projected at the SC15 BoF.]

[Scott: Is accumulation restricted to the use of the $\oplus$ operation of the s argument? If so, add to the specification.]

4.2 Create new descriptor (Descriptor_new)

Creates a new (empty) descriptor.

C99 Syntax

```
#include "GraphBLAS.h"
GrB_info GrB_Descriptor_new(GrB_Descriptor *d)
```

Output Parameters

d Identifier of new descriptor created.

Return Value

GrB_SUCCESS	operation completed successfully
GrB_PANIC	unknown internal error
GrB_OUTOFMEM	not enough memory available for operation
GrB_MISMATCH	mismatch between field and new value

Description

Returns in d the identifier of a newly created empty descriptor. A newly created descriptor can be populated with calls to Descriptor_add.

4.3 Add content to descriptor (Descriptor_add)

[Scott: Naming nit: I propose GrB_Descriptor_set. "adding" implies accumulation (OR) of flags across many calls. Allowing only set which overwrites any existing values is simpler.]

Adds additional content (details of an operation) to an existing descriptor.

C99 Syntax

```
#include "GraphBLAS.h"
GrB_info GrB_Descriptor_add(GrB_Descriptor d,GrB_Field f,GrB_Value v)
```

Input Parameters

d	The descriptor being modified by this method.
f	The field of the descriptor being modified.
v	New value for the field being modified.

Return Value

GrB_SUCCESS	operation completed successfully
GrB_PANIC	unknown internal error
GrB_OUTOFMEM	not enough memory available for operation
GrB_MISMATCH	mismatch between field and new value

Description

The fields of a descriptor include: GrB_OUTP for the output parameter (result) of a method; GrB_MASK for the mask argument to a method; GrB_ARG0 through GrB_ARG9 for the input parameters (from first to last) of a method.

Valid values for a field of a descriptor are as follows:

GrB_NOP	no operation to be performed for the corresponding parameter
GrB_LNOT	compute the logical inverse [Scott: structural complement?] of the corresponding parameter
GrB_TRAN	compute the transpose of the corresponding parameter (for matrices)
GrB_ACC	accumulate result of operation to current values in destination (for output parameter)
GrB_CAST	cast values [Scott: "allow casting of values..."] from input parameters to input domains of operation or from output domain of operation to output parameter. (Otherwise, incorrect domain will cause a run-time error.) [Scott: trying to understand if exact domain match is required or if language type is what we are talking about. Not sure how this would be implemented.]

[Scott: GrB_LNOT clashes with operator in Table 2] [Yzelman: GrB_LNOT: logical inverse of non-mask parameters can be implemented by modifying the operators; therefore consider restricting this to masks only.]

It is possible to specify a combination of values for a field. For example, if a matrix is to be both transposed and logically inverted (element by element), one would use the field value GrB_TRAN | GrB_LNOT.

4.4 Create new space (Space_new)

Creates a new space with specified domain, operations and identities.

C99 Syntax

```
#include "GraphBLAS.h"
GrB_info GrB_Space_new(GrB_Space *s,GrB_type t1, GrB_type t2, GrB_type t3,
                       GrB_operation a,GrB_operation m, t3 z[, t3 o]))
```

[Scott: This signature is a little confusing partially because we have not been explicit with the domain/type specifications earlier. If the GrB_type is the name of an enum for domains (not types) then I would suggest calling it GrB_domain. Not sure how ti can be used as both a type and a parameter and may be related to the domain/type issue.]

Input Parameters

- t1 The type defining the first domain of the space being created. Should be one of the predefined GraphBLAS types in Table 1, or a user created type.
- t2 The type defining the second domain of the space being created. Should be one of the predefined GraphBLAS types in Table 1, or a user created type.
- t3 The type defining the third domain of the space being created. Should be one of the predefined GraphBLAS types in Table 1, or a user created type.
- a The additive operation of the space.
- m The multiplicative operation of the space.
- z The additive identity of the space.
- o The multiplicative identify of the space.

Output Parameter

- s Identifier of the newly created space.

Return Value

GrB_SUCCESS	operation completed successfully
GrB_PANIC	unknown internal error
GrB_OUTOFMEM	not enough memory available for operation

Description

Creates a new space $S = \langle D(\mathsf{t}), \mathsf{a}, \mathsf{m}, \mathsf{z}, \mathsf{o} \rangle$ and returns its identifier in s.

4.5 Create new vector (Vector_new)

Creates a new vector with specified domain and size.

C99 Syntax

```
#include "GraphBLAS.h"
GrB_info GrB_Vector_new(GrB_Vector *v,GrB_type t,GrB_index n)
```

Input Parameters

- t The type defining the domain of the vector being created. Should be one of the predefined GraphBLAS types in Table 1, or a user created type.
- n The size of the vector being created.

Output Parameter

- v Identifier of the newly created vector.

Return Value

GrB_SUCCESS	operation completed successfully
GrB_PANIC	unknown internal error
GrB_OUTOFMEM	not enough memory available for operation

Description

Creates a new vector $\mathbf{v}$ of domain $D(\mathsf{t})$, size n, and empty $L(\mathbf{v})$. It return in v this vector $\mathbf{v}$.

4.6 Number of rows in a matrix (Matrix_nrows)

Retrieve the number of rows in a matrix.

C99 Syntax

```
#include "GraphBLAS.h"
GrB_info GrB_Matrix_nrows(GrB_index *m,GrB_Matrix A)
```

Input Parameters

A Matrix being queried.

Output Parameters

m The number of rows in the matrix.

Return Value

GrB_SUCCESS	operation completed successfully
GrB_PANIC	unknown internal error
GrB_NOMATRIX	matrix does not exist

Description

Return in m the number of rows (parameter M in Section 3.3) in matrix A.

4.7 Assign values to the elements of an object (assign)

[Scott: these variants need to be discussed perhaps in the large group; not currently part of any prior document.]

4.7.1 Flat variant

Set all the elements of a vector to a given value.

```
#include "GraphBLAS.h"
GrB_info GrB_assign(GrB_Vector *v,scalar s[,GrB_Vector m])
```

Input Parameters

- v Vector to be assigned.
- s Scalar value for the elements.
- m (Optional) mask for assignment. [Aydin: Maybe say in the document that GrB_Vector's domain could only be GrB_Index for this function]

Return Value

GrB_SUCCESS	operation completed successfully
GrB_PANIC	unknown internal error
GrB_NOVECTOR	vector does not exist
GrB_MISMATCH	mismatch between vector domain and scalar type

4.7.2 Indexed variant

Set some of the elements of a vector to a given value. [Scott: Set one element of...]

C99 Syntax

```
#include "GraphBLAS.h"
GrB_info GrB_assign(GrB_Vector *v,scalar s,GrB_index i)
```

Input Parameters

- v Vector to be assigned.
- s Scalar value for the elements.
- i Index of element to be assigned

Return Value

GrB_SUCCESS	operation completed successfully
GrB_PANIC	unknown internal error
GrB_NOVECTOR	vector does not exist
GrB_MISMATCH	mismatch between vector domain and scalar type

4.8 Perform of a reduction across the elements of an object (reduce)

Computes the reduction of the values of the elements of a vector or matrix.

4.8.1 Vector variant

C99 Syntax

```
#include "GraphBLAS.h"
GrB_info GrB_reduce(scalar *s,GrB_Vector v,GrB_operations f)
```

[Scott: Should we use the space/algebra in place of the f parameter and just use the $\oplus$ or if an algebra consists of monoids this is another place where a Monoid is appropriate. Note that we must know the identity value for the operation in order to store the correct value in the scalar if the vector that you are reducing has not stored values.]

Input Parameters

- v Vector to be reduced.
- f Operation to be applied in the reduction.

Output Parameters

- s Initial and final value of the reduction.

Return Value

GrB_SUCCESS	operation completed successfully
GrB_PANIC	unknown internal error
GrB_NOVECTOR	vector does not exist
GrB_MISMATCH	mismatch between vector domain and scalar type

4.9 Destroy object (free)

Destroys a previously created GraphBLAS object.

C99 Syntax

```
#include "GraphBLAS.h"
GrB_info GrB_free(GrB_Object o)
```

[Yzelman: polymorphic? is there where _Generic is specified?]

Input Parameter

- GraphBLAS object to be destroyed. Can be a matrix, vector or descriptor.

Return Value

GrB_SUCCESS	operation completed successfully
GrB_PANIC	unknown internal error
GrB_NOOBJECT	object does not exist

[Aydin: add **ewiseadd**and **ewisemult**]

[Scott: add matrix_new, matrix_ncols, matrix_nelts]

A Example: breadth first search with GraphBLAS

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  GrB_info BFS(GrB_Vector *v, GrB_Matrix A, GrB_index s)
8  /*
9   * Given a boolean n x n adjacency matrix A and a source vertex s, performs a BFS traversal
10  * of the graph and sets v[i] to the level in which vertex i is visited (v[s] == 1).
11  * If i is not reacheable from s, then v[i] = 0. (Vector v should be empty on input.)
12  */
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    GrB_assign(v,0);                                 // v = 0
19
20    GrB_Vector q;                                    // vertices visited in each level
21    GrB_Vector_new(&q,GrB_BOOL,n);                   // Vector<bool> q(n)
22    GrB_assign(&q,false);
23    GrB_assign(&q,true,s);                           // q[s] = true, false everywhere else
24
25    GrB_Space Boolean;                               // Boolean space <bool,bool,bool,||,&&,false,true>
26    GrB_Space_new(&Boolean,GrB_BOOL,GrB_BOOL,GrB_BOOL,GrB_LOR,GrB_LAND,false,true);
27
28    GrB_Descriptor desc;                             // Descriptor for vxm
29    GrB_Descriptor_new(&desc);
30    GrB_Descriptor_add(desc,GrB_ARG1,GrB_NOP);       // no operation on the vector
31    GrB_Descriptor_add(desc,GrB_ARG2,GrB_NOP);       // no operation on the matrix
32    GrB_Descriptor_add(desc,GrB_MASK,GrB_LNOT);      // invert the mask
33
34    /*
35     * BFS traversal and label the vertices.
36     */
37    int32_t d = 1;                                   // d = level in BFS traversal
38    bool succ = false;                               // succ == true when some successor found
39    do {
40      GrB_assign(v,d,q);                             // v[q] = d
41      GrB_vxm(&q,Boolean,q,A,*v,desc);               // q[!v] = q ||.&& A ; finds all the unvisited
42                                                     // successors from current q
43      GrB_reduce(&succ,q,GrB_LOR);                   // succ = ||(q)
44      d++;                                           // next level
45    } while (succ);                                  // if there is no successor in q, we are done.
46
47    GrB_free(q);                                     // q vector no longer needed
48    GrB_free(Boolean);                               // Boolean semiring no longer needed
49    GrB_free(desc);                                  // descriptor no longer needed
50
51    return GrB_SUCCESS;
52 }
```

B Example: betweenness centrality with GraphBLAS

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  GrB_info BC(GrB_Vector *delta, GrB_Matrix A, GrB_index s)
8  /*
9   * Given a boolean n x n adjacency matrix A and a source vertex s,
10  * compute the BC-metric vector delta, which should be empty on input.
11  */
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    GrB_assign(delta, 0);                           // delta = 0
18
19    GrB_Matrix sigma;                               // Matrix<int32_t> sigma(n,n)
20    GrB_Matrix_new(&sigma, GrB_INT32, n, n);        // sigma[d,k] = shortest path count to node k
21    GrB_assign(&sigma, 0);                          // at level d
22
23    GrB_Vector q;
24    GrB_Vector_new(&q, GrB_INT32, n);               // Vector<int32_t> q(n) of path counts
25    GrB_assign(&q, 0);                              // q = 0
26    GrB_assign(&q, 1, s);                           // q[s] = 1
27
28    GrB_Vector p;
29    GrB_Vector_new(&p, GrB_INT32, n);               // Vector<int32_t> p(n) shortest path counts so far
30    GrB_assign(&p, q);                              // p = q
31
32    GrB_Space Int32;                                // Space <int32_t,int32_t,int32_t,+,*,0,1>
33    GrB_Space_new(&Int32,GrB_INT32,GrB_INT32,GrB_INT32,GrB_PLUS,GrB_TIMES,0,1);
34
35    GrB_Descriptor desc;                            // Descriptor for vxm
36    GrB_Descriptor_new(&desc);
37    GrB_Descriptor_add(desc,GrB_ARG1,GrB_NOP);      // no operation on the vector
38    GrB_Descriptor_add(desc,GrB_ARG2,GrB_NOP);      // no operation on the matrix
39    GrB_Descriptor_add(desc,GrB_MASK,GrB_LNOT);     // invert the mask
40
41    /*
42     * BFS phase
43     */
44    int32_t d = 0;                                  // BFS level number
45    int32_t sum = 0;                                // sum == 0 when BFS phase is complete
46    do {
47       GrB_assign(&sigma,q,d,GrB_ALL);              // sigma[d,:] = q
48       GrB_vxm(&q,Int32,q,A,p,desc);                // q = # paths to nodes reachable from current level
49       GrB_ewiseadd(&p,Int32,p,q);                  // accumulate path counts on this level
50       GrB_reduce(&sum,q,GrB_PLUS);                 // sum path counts at this level
51       d++;
52    } while (sum);
53
54    /*
55     * BC computation phase
56     * (t1,t2,t3,t4) are temporary vectors
57     */
58    GrB_Space FP32AddMul;                           // Space <float,float,float,+,*,0,1>
59    GrB_Space_new(&FP32AddMul,GrB_FP32,GrB_FP32,GrB_FP32,GrB_PLUS,GrB_TIMES,0.0,1.0);
60
61    GrB_Space FP32AddDiv;                           // Space <float,float,float,+,/,0,1>
62    GrB_Space_new(&FP32AddDiv,GrB_FP32,GrB_FP32,GrB_FP32,GrB_PLUS,GrB_DIV,0.0,1.0);
```

```
63
64    GrB_Vector t1; GrB_Vector_new(&t1,GrB_FP32,n);
65    GrB_Vector t2; GrB_Vector_new(&t2,GrB_FP32,n);
66    GrB_Vector t3; GrB_Vector_new(&t3,GrB_FP32,n);
67    GrB_Vector t4; GrB_Vector_new(&t4,GrB_FP32,n);
68    for(int i=d-1; i>0; i--)
69    {
70      GrB_assign(&t1, 1);                          // t1 = 1+delta
71      GrB_ewiseadd(&t1,FP32AddMul,t1,*delta);
72      GrB_assign(&t2,sigma,i,GrB_ALL);             // t2 = sigma[i,:]
73      GrB_ewisemul(&t2,FP32AddDiv,t1,t2);          // t2 = (1+delta)/sigma[i,:]
74      GrB_mxv(&t3,FP32AddMul,A,t2);                // add contributions made by successors of a node
75      GrB_assign(&t4,sigma,i-1,GrB_ALL);           // t4 = sigma[i-1,:]
76      GrB_ewisemul(&t4,FP32AddMul,t4,t3);          // t4 = sigma[i-1,:]*t3
77      GrB_ewiseadd(delta,FP32AddMul,*delta,t4);    // accumulate into delta
78    }
79
80    GrB_free(q); GrB_free(p);
81    GrB_free(Int32); GrB_free(FP32AddMul); GrB_free(FP32AddDiv);
82    GrB_free(desc);
83
84    return GrB_SUCCESS;
85  }
```