**The Standard Versions of extract and assign (from the “args” spreadsheet):**

Currently we do not have the “standard” version of either operation as defined by the math document and further refined by the spreadsheet that lists the set of arguments including which operands can be transposed and which can be negated. The following is the matrix version:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \*extract\* | **C** ⊕= ¬**A**T(**i**,**j**) | **C** | ⊕= | ¬ | **A** | T | **i** | **j** |
| \*assign\* | **C**(**i**,**j**) ⊕= ¬**A**T | **C** | ⊕= | ¬ | **A** | T | **i** | **j** |
| Argname type |  | dst Matrix | accum Function | negate (Descriptor) | src Matrix | transpose (Descriptor) | row Index Array | col Index Array |

We need to define the vector version similarly:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| \*extract\* | **c** ⊕= ¬**a**T(**i**) | **c** | ⊕= | ¬ | **a** | T | **i** |
| \*assign\* | **c**(**i**) ⊕= ¬**a**T | **c** | ⊕= | ¬ | **a** | T | **i** |

* Should we support masking? I believe the answer is yes based on some observations below.

\*\*\* Aydin’s note for MATLAB programmers \*\*\*

The logic here is that we want to support Matlab style individual element indexing in both r- and l-value. Currently, the API is such that it only supports A(i,j) style coarse-grained indexing. To achieve A(M) type indexing where both A and M are n-by-m matrices, here we use the Mask concept. This also covers the output mask, which would have implemented in Matlab via A(M) = B.

My (updated) understanding any ‘assignment’ to a dst matrix/vector is that if dst was not empty prior then it behaves “as if” all previous contents of dst are cleared out prior to the assignment. I don’t know why I thought existing language in either the math or API documents was ambiguous on this point. At some point I got the impression that assignment could only store new elements and not remove existing elements in the destination.

* If anyone else had the same misconceptions we need to strengthen the wording to clarify the correct behavior in the Description sections (I only have vxm documentation to refer to right now); otherwise it is only me.

Then, I examined what assigns are used in the code examples and my attempt to define and possibly unify some of what they do:

Accessing single values

* **BFS, line 23: GrB\_assign(&q, true, s);**
* **BC, line 26: GrB\_assign(&q, 1, s);**
  + Signature: GrB\_assign(GrB\_Vector \*dst, scalar value, GrB\_Index i);
  + This stores a single element in a vector (a matrix version would have two indices): q[s] = <some value>
  + This would be a convenience function for not having to create IndexArrays holding one index and an array holding one scalar.
  + Since there is no context of the semiring, it cannot be used to remove a stored value (no way to detect additive identity).
  + Is this convenience function worth it? I believe the answer is yes as it happens often enough in the set up phase of many algorithms.
  + When trying to create an extract version that mirrors this, I run into issues when attempting to access a structural zero. If dst is a pointer to a scalar, then I need information from the semiring about the value to return. Note the standard version with sparse mat/vec does not have this problem because you can return an empty vector/matrix of size 1/1x1.

Clearing or Filling

* **BFS, line 18: GrB\_assign(v, 0);**
* **BFS, line 22: GrB\_assign(&q, false);**
* **BC, lines 17, 21, 25**…same thing
* **BC, line 79: GrB\_assign(&t1, 1);**
  + Signature: GrB\_assign(GrB\_Vector \*dst, scalar value);
  + Generally speaking this function performs a “fill” of the value passed.
  + The intent in these specific uses are to clear the sparse vectors, and since they immediately follow creation, they can be removed.
  + Even though the intent was to clear sparse vectors, knowledge of the semiring identities is required to determine this (otherwise we fill them with the value that happens to be the additive identity)
  + I believe sparse Vectors (and Matrices) should have a “clear” method instead and specifying the value is not necessary for sparse vectors and matrix types either. Dense versions are a different story as we would need the “fill” version to accomplish this behavior.
  + If Tim gets his way, Semirings would be attached to each matrix or vector and clear could use the info in the semiring to determine what to do if necessary. So far the implementation I have gets away with just storing the additive identity with the Matrices.
* **BFS, line 40: GrB\_assign(v, d, q);**
  + Signature: GrB\_assign(Vector \*dst, scalar value, const Vector mask)
    - dst and mask (v and q) must have the same dimensions
    - In this description I am generalizing Jose’s use where the mask (q) vector is specifically Boolean. In this use, everywhere mask (a sparse vector) has a stored value (not specifically ‘true’), the corresponding element in v (dst) gets a stored value equal to a constant ‘value’.
  + This is actually a generalization of the “fill” function by adding the mask.
  + Is this the same as providing an IndexArray instead of a mask?
    - I believe the answer depends on the behavior of masked out elements verses indices not present in the IndexArray. Where the elements of mask have a structural zero, I assume this means that corresponding dst locations should be “cleared”. This is not the case in the IndexArray version where non-references locations are left alone.
    - If that is true then line 40 is a convenience function for two calls to the IndexArray version
  + In the case of dense, knowledge of two additive identities is needed to (1) resolve the mask and (2) to know what to clear dst locations to (in the dense case).

Assignment operator or copy construction

* **BC, line 30: GrB\_assign(&p, q); // p := q**
  + Signature: GrB\_assign(GrB\_Vector \*dst, GrB\_Vector src)
  + My understanding: create a duplicate of q
  + One possibility is there should be a variant of Matrix\_new and Vector\_new that create copies of existing matrices and vectors (that would be applicable in this particular case), but do we also need the equivalent of an assignment operator?
  + The other way to look at this: a convenience function for extractTuples + buildMatrix.
  + Do we need to discuss what distinguishes extract/assign and extractTuples/buildMatrix?

Extracting and assigning rows and columns of the Matrix

* **BC, line 50: GrB\_assign(&sigma,q,d,GrB\_ALL); // sigma[d,:] = q**
  + Signature: GrB\_assign(GrB\_Matrix \*dst, GrB\_Vector const src, GrB\_Index row\_index, GrB\_IndexArray col\_indices)
    - I am not really sure what the last argument is but I can convince myself that in the general case it is in IndexArray like in the standard case
    - Is the GrB\_ALL a token indicating the concept of an IndexArray with a “complete” set of indices {0..n-1}?
  + This replaces a row of the dst matrix by setting all locations in row to the src vector
  + Is there a column version?:
    - GrB\_assign(GrB\_Matrix \*dst, GrB\_Vector const src, GrB\_IndexArray row\_indices, GrB\_Index col\_index)
    - I don’t think you can do this using “transpose” with the first version as we do not currently support transpose of dst.
  + Do we need to worry about orientation of src vector versus orientation of row/col being modified?
* **BC, line 81: GrB\_assign(&t2,sigma,i,GrB\_ALL); // t2 = sigma[i,:]**
* **BC, line 84: GrB\_assign(&t4,sigma,i-1,GrB\_ALL); // t4 = sigma[i-1,:]**
  + Signature: GrB\_assign(GrB\_Vector \*dst, GrB\_Matrix const src, GrB\_Index row\_index, GrB\_IndexArray col\_indices)
  + This is the “reverse” of the previous (extracting a row from matrix and assigning to a vector)…this should be extract
  + One possible issue is the assignment of a row vector to a column vector. We have not discussed what is and is not allow with regard to the orientation of vectors.
    - In this case transpose of the Matrix could be used to extract both rows and columns from the matrix and resolve any orientation issues that may arise from the assignment to a destination column vector.

Standard versions

GrB\_info GrB\_assign(GrB\_Vector \*dst,

GrB\_Function const accum,

GrB\_Vector const src,

GrB\_IndexArray const rows

[, GrB\_Vector const mask

[, GrB\_Descriptor const desc]]);

GrB\_info GrB\_assign(GrB\_Matrix \*dst,

GrB\_Function const accum,

GrB\_Matrix const src,

GrB\_IndexArray const rows,

GrB\_IndexArray const cols

[, GrB\_Matrix const mask

[, GrB\_Descriptor const desc]]);

GrB\_info GrB\_extract(GrB\_Vector \*dst,

GrB\_Function const accum,

GrB\_Vector const src,

GrB\_IndexArray const rows

[, GrB\_Vector const mask

[, GrB\_Descriptor const desc]]);

GrB\_info GrB\_extract(GrB\_Matrix \*dst,

GrB\_Function const accum,

GrB\_Matrix const src,

GrB\_IndexArray const rows,

GrB\_IndexArray const cols

[, GrB\_Matrix const mask

[, GrB\_Descriptor const desc]]);

Notes:

* Any GrB\_IndexArray can be specified as GrB\_ALL which indicates the set all of the valid index values in that dimension (unpermuted).

Row and Column variants

GrB\_info GrB\_assign(GrB\_Matrix \*dst,

GrB\_Function const accum,

GrB\_Vector const src,

GrB\_Index row, // assign row

GrB\_IndexArray const cols

[, GrB\_Matrix const mask

[, GrB\_Descriptor const desc]]);

GrB\_info GrB\_assign(GrB\_Matrix \*dst,

GrB\_Function const accum,

GrB\_Vector const src,

GrB\_IndexArray const rows

GrB\_Index col // assign col

[, GrB\_Matrix const mask

[, GrB\_Descriptor const desc]]);

GrB\_info GrB\_extract(GrB\_Vector \*dst,

GrB\_Function const accum,

GrB\_Matrix const src,

GrB\_Index row, // extract row

GrB\_IndexArray const cols

[, GrB\_Vector const mask

[, GrB\_Descriptor const desc]]);

GrB\_info GrB\_extract(GrB\_Vector \*dst,

GrB\_Function const accum,

GrB\_Matrix const src,

GrB\_IndexArray const rows

GrB\_Index col // assign col

[, GrB\_Vector const mask

[, GrB\_Descriptor const desc]]);

Notes:

* Pulling a row or column out of a Matrix is extraction of a subgraph
* Assigning a row or column in a Matrix is assignment of a subgraph
* Reminder: Any GrB\_IndexArray can be specified as GrB\_ALL which indicates the set all of the valid index values in that dimension (unpermuted).
* Should the mask type be tied to the dst type?

Single-element variants

GrB\_info GrB\_assign(GrB\_Vector \*dst,

GrB\_Function const accum,

scalar src,

GrB\_Index row

[, GrB\_Descriptor const desc]);

GrB\_info GrB\_assign(GrB\_Matrix \*dst,

GrB\_Function const accum,

scalar src,

GrB\_Index row,

GrB\_Index col

[, GrB\_Descriptor const desc]);

GrB\_info GrB\_extract(scalar \*dst,

GrB\_Function const accum,

GrB\_Vector const src,

GrB\_Index row

[, GrB\_Descriptor const desc]);

GrB\_info GrB\_extract(scalar \*dst,

GrB\_Function const accum,

GrB\_Matrix const src,

GrB\_Index row,

GrB\_Index col

[, GrB\_Descriptor const desc]);

Notes:

* Masking not necessary,
* TODO: what Descriptor options need to be supported?
* TODO: How to handle extract access to a structural zero from a sparse src? Need to know additive identity.

Constant/Filling assign

GrB\_info GrB\_assign(GrB\_Vector \*dst,

GrB\_Function const accum,

scalar src,

GrB\_IndexArray rows

[, GrB\_Vector const mask

[, GrB\_Descriptor const desc]]);

GrB\_info GrB\_assign(GrB\_Matrix \*dst,

GrB\_Function const accum,

scalar src,

GrB\_IndexArray rows,

GrB\_IndexArray cols

[, GrB\_Matrix const mask

[, GrB\_Descriptor const desc]]);

Notes:

* Cannot accomplish a clear without having information about additive identity?
* There is no corresponding filling extract functionality (the constant is not a graph therefore you cannot extract a subgraph from it.
* Reminder: Any GrB\_IndexArray can be specified as GrB\_ALL which indicates the set all of the valid index values in that dimension (unpermuted).
* What is the semantic using the mask?