Named and defined types and operators

Tim Davis
davis@tamu.edu, Texas A&M University.

December 29, 2021

Abstract

GraphBLAS needs a mechanism for providing names (as strings) to its types and operators both built-in and user-defined. These strings can then be returned by a method that queries the type of a matrix. Closely related is a need to define types and operators using strings, so that these strings can be interpretted by a JIT compiler.

1 The problems

1.1 Cannot query the type of a matrix

There currently is a fundamental difficulty in writing algorithms using GraphBLAS. It is impossible to query the type of a `GrB_Matrix`, `GrB_Vector`, or `GrB_Scalar`. This makes it hard to write an algorithm that depends on the types of its inputs.

For example, consider a shortest-path algorithm. It would be given an adjacency matrix of any type, and it would like to use the operators and semirings that are approriate for this type. This is impossible with the v2.0 C API.

The problem is "solved" in LAGraph by including the matrix type in the `LAGraph_Graph` data structure:

```
struct LAGraph_Graph_struct
{
    GrB_Matrix   A;          // the adjacency matrix of the graph
    GrB_Type     A_type;     // the type of scalar stored in A
    ...
} ;
```

Where `A_type` is the type of the matrix `A`. However, this is clumsy and error-prone. It duplicates information that is already available inside the matrix `A`, just inaccessible. If the type of `A` and the variable `A_type` do not match, bad things can happen inside the LAGraph algorithms.

We thus need a mechanism where the user application can query a `GrB_Matrix`, `GrB_Vector`, or `GrB_Scalar` for its type.

The argument: "The user should know what types its matrices are" is not a good argument against the need for this type query. The matrix dimension should be known to the user too, but the GraphBLAS C API spec provides this information with `GrB_Matrix_nrows` anyway.

Furthermore, the user might know the matrix type, but a library such as LAGraph, that resides in the layer between the user application and the underlying GraphBLAS library, may not know the type, as in the shortest-path algorithm example.

Should a library such as LAGraph ask this of the end user?

> *Please pass me your matrix, so I can run some cool algorithm on it for you. Oh, and tell me the type of the matrix because I have*

> *no clue and can't ask GraphBLAS to tell me. By the way, don't get it wrong or else I'll segfault when doing* `GrB_extractTuples` *to an array whose size depends on the size of this unqueriable type. Good luck.*

Surely not.

1.2 Cannot query the type of a serialized "blob"

Related to Section 1.1 is the problem with the serialization/deserialization of a `GrB_Matrix` to/from a "blob" of bytes (the spec doesn't call it a blob but that's not relevant for this discussion).

The current methods in the v2.0 C API are asymmetric, and need to be so:

```
GrB_Info GrB_Matrix_serialize       // serialize a GrB_Matrix to a blob
(
    // output:
    void *blob,                     // the blob, already allocated in input
    // input/output:
    GrB_Index *blob_size_handle,    // size of the blob on input.  On output,
                                    // the # of bytes used in the blob.
    // input:
    GrB_Matrix A                    // matrix to serialize
) ;

GrB_Info GrB_Matrix_deserialize     // deserialize blob into a GrB_Matrix
(
    // output:
    GrB_Matrix *C,      // output matrix created from the blob
    // input:
    GrB_Type type,      // type of the matrix C
    const void *blob,       // the blob
    GrB_Index blob_size     // size of the blob
) ;
```

When a matrix is serialized, the underlying library already knows its `GrB_Type`, so this does not appear in the signature for the `GrB_Matrix_serialize` method.

Built-in types can be encoded into the `blob`, and this could be reconstructed when the matrix is deserialized. User-defined types are difficult. All the GraphBLAS library knows about a user-defined type is its size in bytes. Even though the library knows the matrix has a specific user-defined `GrB_Type`, the library cannot save this information to the `blob` directly. The `GrB_Type` is an ephemeral pointer, and it goes away when the process finishes.

The purpose of serialization/deserialization is to create a blob of bytes that can be saved to a file, sent to another MPI process running the same GraphBLAS library, or some other method that goes outside the scope of the process creating the blob. The `GrB_Type` for user-defined types is just a pointer to a `malloc`ed block. It cannot be sent to another process and cannot be saved to a file.

In SuiteSparse:GraphBLAS, I extend the usage of `GrB_Matrix_deserialize` slightly, to make it easier to deserialize a matrix of built-in type. I state in my user guide the `GrB_Type type` input to `GrB_Matrix_deserialize`: *Required if the blob holds a matrix of user-defined type. May be NULL if blob holds a built-in type; otherwise must match the type of C.*

However, this does not solve the problem for user-defined types.

Suppose a user writes a blob to a file with a user-defined type:

```
typedef struct
{
    float stuff [4][4] ;
    char whatstuff [64] ;
}
wildtype ;                      // C version of wildtype

GrB_Type WildType ;             // GraphBLAS version of wildtype
...
GrB_Type_new (&WildType, sizeof (wildtype)) ;
GrB_Matrix_new (&A, WildType, ...) ;
...
GrB_Matrix_serialize (blob, &blobsize, A) ;
FILE *f = fopen ("mystuff.bin", "w") ;
fwrite (&blobsize, sizeof (GrB_Index), 1, f) ;
fwrite (blob, sizeof (char), blobsize, f) ;
fclose (f) ;
```

This is not a good use of `GrB_Matrix_serialize`. When the file is reopened, the user must magically know that the

```
FILE *f = fopen ("mystuff.bin", "r") ;
GrB_Index blobsize ;
fread (&blobsize, sizeof (GrB_Index), 1, f) ;
void *blob = malloc (blobsize) ;
```

```
fread (blob, sizeof (char), blobsize, f) ;
fclose (f) ;
GrB_Matrix_deserialize (&A, WildType, blob, blobsize) ;
```

This solution is fragile. What if the wrong type is passed to deserialize? Worse, what if the type has the wrong size? A segfault or security problem is waiting to happen, particularly if the file can from another source.

Yet the GraphBLAS library can provide little help here. The library does not know the name of the type (neither the `typedef struct` name, `wildtype`, nor the `GrB_Type WildType`). All it knows is the size, in this case probably 128 bytes. The library could save in the blob some indicator that the type is user-defined, of size 128 bytes. However, the library has no way of communicating this information to the user application.

2 Difficulties in the solution

Consider the simple case of querying a `GrB_Matrix A` for its type. Should this return a pointer to a `GrB_Type t`? If so, how long does this variable `t` last? And who owns it; is it part of the matrix `A`? Or is it owned by the user application, who must at some point free the variable `t`?

Consider the deserialization case. Querying the blob for its type and returning the type as a `GrB_Type t` is not a good solution. That is possible if the type is built-in, but not possible for user-defined types. The GraphBLAS library has no knowledge of the user-defined type except the pointer itself (which cannot be saved in the blob) and its size in bytes, which would be helpful to know but is not sufficient to know the actual type: `WildType`, constructed from the `typedef wildtype`.

Furthermore, type information cannot be passed between MPI processes. One process cannot send a `GrB_Type` pointer to another process.

As a result of all these issues, return a `GrB_Type` for type queries is not a good idea.

3 The solution: named types (using strings)

Give each type a name as a string, including built-in and user-defined, and allow the type of a matrix to be querried by returning this string.

3.1 A revised GrB_Type_new method

The name of a built-in type should match the underlying C type, but held as a string. For example, the name of the GrB_BOOL type should be "bool", GrB_INT32 is "int32_t", and GrB_FP32 is "float", and so on. All types are named with strings.

Furthermore, the GraphBLAS library should know more about user-defined types than just their name (as a string) and size (in bytes). It should know the entire typedef, as a string, so it can employ JIT acceleration for user-defined types.

SuiteSparse:GraphBLAS includes the following GxB extension:

```
// GxB_Type_new creates a type with a name and definition that are known to
// GraphBLAS, as strings.  The type_name is any valid string (max length of 128
// characters, including the required null-terminating character) that may
// appear as the name of a C type created by a C "typedef" statement.  It must
// not contain any white-space characters.  Example, creating a type of size
// 16*4+1 = 65 bytes, with a 4-by-4 dense float array and a 32-bit integer:
//
//      typedef struct { float x [4][4] ; int color ; } myquaternion ;
//      GrB_Type MyQtype ;
//      GxB_Type_new (&MyQtype, sizeof (myquaternion), "myquaternion",
//          "typedef struct { float x [4][4] ; int color ; } myquaternion ;") ;
//
// The type_name and type_defn are both null-terminated strings.  Currently,
// type_defn is unused, but it will be required for best performance when a JIT
// is implemented in SuiteSparse:GraphBLAS (both on the CPU and GPU).  User
// defined types created by GrB_Type_new will not work with a JIT.
//
// At most GxB_MAX_NAME_LEN characters are accessed in type_name; characters
// beyond that limit are silently ignored.

#define GxB_MAX_NAME_LEN 128

GrB_Info GxB_Type_new           // create a new named GraphBLAS type
(
    GrB_Type *type,             // handle of user type to create
    size_t sizeof_ctype,        // size = sizeof (ctype) of the C type
    const char *type_name,      // name of the type (max 128 characters)
    const char *type_defn       // typedef for the type (no max length)
) ;
```

The only downside to this solution is that the typedef is repeated twice: once in the C program to define the type myquaterion, in this exmpale, and again in the string passed to GxB_Type_new. This duplication could be avoided with C macros.

3.2 A new GrB_Matrix_type_name method

Since all types, including built-in types, are given a name, and since this name does not exceed 128 bytes in length, the user can query the type of a matrix with the following method:

```
GrB_Info GxB_Matrix_type_name      // return the name of the type of a matrix
(
    char *type_name,        // name of the type (char array of size at least
                            // GxB_MAX_NAME_LEN, owned by the user application).
    const GrB_Matrix A      // matrix to query
) ;
```

Usage:

```
char typename [GxB_MAX_NAME_LEN] ;
GxB_Matrix_type_name (typename, A) ;
```

If the matrix has type GrB_FP64, the typename now contains the null-terminated string "double". If the matrix has type WildType or MyQtype, then the typename array holds the null-terminated string wildtype or myquaternion, respectively.

Ownership and lifetime of the return value from GxB_Matrix_type_name is well-defined. The result is copied into a user-owned char array. The user application has full control and ownership of this array, and the array typename can outlive the existence of the matrix A and/or its type, WildType.

3.3 A new GrB_Type_size method

Methods such as GrB_Matrix_extractTuples require a user array, call it X passed to it. This array must be able to hold nvals entries, each of size equal to the number of bytes needed to hold the size of the matrix type.

GrB_Matrix_nvals can report the number of values.

What is the type of the matrix? And what is the size of this type, so the user can allocate the proper sized array? GraphBLAS cannot tell you, with the current v2.0 C API. There is a need for querying the size of a type, so that GxB_Type_size (&s, GrB_FP32) would return sizeof(float), or typically 4, for example.

```
GrB_Info GxB_Type_size          // determine the size of the type
(
    size_t *size,               // the sizeof the type
    const GrB_Type type         // type to determine the sizeof
) ;
```

3.4 A new GrB_Matrix_serialized_type_name method

Now that the type has a name, as a string, the string can be safely inserted by the library into a serialized blob. Then, when the blob is reloaded from a file, or received from another MPI process, the string can be safely parsed and returned to the user application:

```
GrB_Info GxB_deserialize_type_name  // return the type name of a blob
(
    // output:
    char *type_name,        // name of the type (char array of size at least
                            // GxB_MAX_NAME_LEN, owned by the user application).
    // input, not modified:
    const void *blob,       // the blob
    GrB_Index blob_size     // size of the blob
) ;
```

Usage: reading a blob from a file and getting its type

```
FILE *f = fopen ("mystuff.bin", "r") ;
GrB_Index blobsize ;
fread (&blobsize, sizeof (GrB_Index), 1, f) ;
void *blob = malloc (blobsize) ;
fread (blob, sizeof (char), blobsize, f) ;
fclose (f) ;

// get the typename of the matrix held in the blob:
char typename [GxB_MAX_NAME_LEN] ;
GxB_deserialized_type_name (typename, blob, blob_size) ;

if (strcmp (typename, "wildtype"))
{
    GrB_Matrix_deserialize (&A, WildType, blob, blobsize) ;
}
else if (strcmp (typename, "myquaternion"))
{
    GrB_Matrix_deserialize (&A, MyQType, blob, blobsize) ;
}
```

```
    else if (strcmp (typename, "float"))
    {
        GrB_Matrix_deserialize (&A, GrB_FP32, blob, blobsize) ;
    }
        ... and so on ...
```

Better yet, if the blob has a built-in type, the library can handle this itself. This can work for a blob of a matrix of any built-in type:

```
    GrB_Matrix_deserialize (&A, NULL, blob, blobsize) ;
```

Then the user application need only compare the `typename` string for its known user-defined types, and after that, assume the blob has a built-in type.

Since the library now knows the name of the type of the matrix held in the blob, it can know if the name is a recognized built-in type (`"bool"`, `"int8_t"`, `"float"`, ... `"double"`). If the `type` input parameter to `GrB_Matrix_deserialize` is `NULL`, and the blob has a known built-in type, it can handle this itself. The library can also record the name and size in bytes of a user-defined type, and if the `type` parameter is `NULL`, it can refuse to deserialize it, since it nows the blob has a matrix of user-defined type.

3.5 A new method to convert the string to the type

Eventually, there would be a need to convert the `typename` string to an actual `GrB_Type`. This can easily be done by the GraphBLAS library for built-in types. For user-defined types, I suggest that this function return `GrB_NO_VALUE` and a `NULL` type, so the user application can convert the string to a type.

```
GrB_Info GxB_Type_from_name     // return the built-in GrB_Type from a name
(
    GrB_Type *type,             // built-in type, or NULL if user-defined
    const char *type_name       // array of size at least GxB_MAX_NAME_LEN
) ;
```

Usage:

```
    // to query the type of A:
    size_t typesize ;
    GxB_Type_size (&typesize, type) ;       // works for any type
```

```
GrB_Type atype ;
char atype_name [GxB_MAX_NAME_LEN] ;
GxB_Matrix_type_name (atype_name, A) ;
GxB_Type_from_name (&atype, atype_name) ;
if (atype == NULL)
{
    // This is not yet an error.  It means that A has a user-defined type.
    if ((strcmp (atype_name, "myfirsttype")) == 0) atype = MyType1 ;
    else if ((strcmp (atype_name, "myquaternion")) == 0) atype = MyQType ;
    else if ((strcmp (atype_name, "wildetype")) == 0) atype = WildType ;
    else { ... this is now an error ... the type of A is unknown.  }
    }
}
```

4 JIT acceleration

4.1 The problem: JIT acceleration is impossible with the current API

With types named with a string as `myquaternion` for example, and also defined with a string as this for example:

```
"typedef struct { float x [4][4] ; int color ; } myquaternion ;"
```

the GraphBLAS library has powerful tools to reason about the types of its matrices. It can provide this information back to the user application, through safe type queries. It can use this information to make serialization/deserialization safer and more reliable. It can allow LAGraph to ditch its clumsy solution of passing along the type of a matrix with the matrix itself.

However, this is only a partial solution in how a GraphBLAS library could deal with user-defined types. These UDT's also require operators. Currently, all a GraphBLAS library knows about an operator is a function pointer. It doesn't even know the name of these operators, just a pointer. This limits the performance of a GraphBLAS library when working on user-defined types.

The GraphBLAS library may wish to construct fast kernels at run time for user-defined types and operators. This requies the library to open a new file, write a C program into it (with these user-defined types and operators), compile the function, link it in, and then call the kernel. The next time this kernel is called, it can keep a record of it, and not reconstruct and recompile it but just use the existing linked-in compiled code.

This is essential on the GPU, since the GPU cannot call a function pointer defined on the CPU and passed to GrB_BinaryOp_new.

4.2 The solution: give names and definitions to all user-defined operators

Consider the following extension:

```
GrB_Info GxB_BinaryOp_new
(
    GrB_BinaryOp *op,               // handle for the new binary operator
    GxB_binary_function function,   // pointer to the binary function
    GrB_Type ztype,                 // type of output z
    GrB_Type xtype,                 // type of input x
    GrB_Type ytype,                 // type of input y
    const char *binop_name,         // name of the user function
    const char *binop_defn          // definition of the user function
) ;
```

This method augments GrB_BinaryOp_new by giving the new operator a name, as a string (of length GxB_MAX_NAME_LEN) and by giving the definition of the entire function itself. For example, suppose the user wanted to define a mod operator that worked on uint64_t types. The current solution is limited:

```
    void mod_function (void *z, const void *x, const void *y)
    {
        uint64_t a = (*((uint64_t *) x)) ;
        uint64_t b = (*((uint64_t *) y)) ;
        (*((uint64_t *) z)) = a % b ;
    }
    ...
    GrB_BinaryOp Mod ;
    GrB_BinaryOp_new (&Mod, mod_function, GrB_UINT64, GrB_UINT64, GrB_UINT64)) ;
```

Consider the information that the GraphBLAS library knows about this new operator: it has an ephemeral pointer, Mod, to some mallocd block of memory that holds the GrB_BinaryOp object. This object has three pointers to three GrB_Type objects. These are built-in types, so the library knows that they correspond to the C uint64_t type. If the type was user-defined, unless the `GxB_Type_new` is used (Section 3.1), all it knows is the size of these user-defined types.

The library has no idea of the contents of the `mod_function`. This function pointer is compiled for use on the CPU. It cannot be called from the GPU, so GPU acceleration of this user-defined function is impossible.

Now consider a better alternative:

```
GxB_BinaryOp_new (&Mod, mod_function, GrB_UINT64, GrB_UINT64, GrB_UINT64
    "mod_function",
    "void mod_function (uint64_t *z, const uint64_t *x, const uint64_t *y)"
    "{"
    "    z = x % y ;"
    "}") ;
```

First, the parameters no longer need to be `void` and require typecasting. Typecasting a pointer from `void *` to `uint64_t *` requires no work at run time, just some work by the compiler, but it makes the code harder to read. Now, however, the library knows that the types have the C name of `uint64_t` so it can use those directly.

For a JIT on the CPU, the GraphBLAS library could create a short file like this. The library it knows the name (`"mod_function"`) and so it can build a full name of a kernel, and it knows the definition of the `mod_function` itself, as a string, so it can build a static inline copy of it:

```
static inline
    void mod_function (uint64_t *z, const uint64_t *x, const uint64_t *y)
    {
        z = x % y ;
    }
#define OP(z,x,y) mod_function (&z, &x, &y)
#define GB_EWISEADD_KERNEL_NAME GB_ewiseAdd_kernel_for_mod_function
#include "GB_eWiseAdd_kernel.c"
```

where the file `GB_eWiseAdd_kernel.c` would be part of the library, like my `Source/Template/*.c` files in SuiteSparse:GraphBLAS. Those kernels work with any data type and any operator, where the types and operators are `#define`d by C macros.

I have not implemented this, but it is feasible. I have implemented kernels like this for the GPU, but only for built-in types and operators.

This kernel would be far faster than computing the ewiseadd operation by calling the function pointer every time. Function pointers are very slow.

4.3 Named and defined monoids and semirings

Named monoids are not essential, since their name can be constructed from the name of the underlying binary operator, whether user-defined or built-in. However, the monoid identity value is likely needed as a string, since the GraphBLAS library would otherwise have difficulty in constructing a string that defines it an placing that string in a file to be compiled.

A semiring is even simpler. It consists of just a binary multiplicative operator, and a monoid. The name of the semiring can be built from the names of these two objects, as well as the names of the types they operate on (up to 3 of them for the multiplier and monoid, plus 3 more if the types of the 3 matrices are also considered). All of these objects would have named and defined types and operators that underly them, so an entire semiring could be encoded in a string, placed in a file, compiled, linked, and executed.

5 Summary

Use strings to give all types and operators a name. The name should be short, of some fixed maximum length.

Use the strings to allow all types to be queried, for all objects:

1. What is the type of a `GrB_Matrix`?
2. What is the type of a `GrB_Vector`?
3. What is the type of a `GrB_Scalar`?
4. What is the type of a matrix held inside a serialized blob?
5. What is the type of the y input to a binary operator $f = (x, y)$?
6. etc.

Use strings to give all types and operators a precise definition. The definition cannot be limited by a maximum length. These definitions can then be used by a JIT on both the CPU and the GPU, so that user-defined types and operators can be just as fast as built-in ones.