Conversion Between Text and Datatype

Quincey Koziol & Raymond Lu
koziol@ncsa.uiuc.edu, slu@ncsa.uiuc.edu

Aug 11, 2004
Revised on Sep 21, 2004

I. Document's Audience

Current HDF5 library designers and knowledgeable external developers.

II. Requirements and Use Cases

1. There have been some user requests to create HDF5 data type in a single step. Although it does not save much for atomic data data type creation into one step for some more complex data types, like compound and array types. The ASCI people have been t a data type text description in order to define a data type during run time in a single place.

2. On the other side, some user requests the text description of a HDF5 data type for debugging purpose. This description can be is quite similar to what the `h5dump` does.

3. Another request is to make a HDF5 private function `H5T_cmp` public. What it does is to compare data types in the library's pre table of data types.

III. Functionality

There are two new major functions we are going to add to the library to satisfy the requirements, one is `H5Ttext_to_type`, anothe on the format of text description for a data type, `H5Ttext_to_type` converts a text description into an HDF5 datatype object and r `H5Ttype_to_text`, it returns a text description of an HDF5 datatype given by its ID. The format of the text should comply with t have predefined. More details of these two functions can be found in *Section VIII* of this document.

There will be another function called `H5Tcmp`. It compares two data types in the arbitrary way of the library's predefinition.

IV. Library Design

1. Conversion from text to data type.

For the internal library design, there are several steps involved to convert from text description of a data type to an HDF5 data type illustrated in the following diagram,

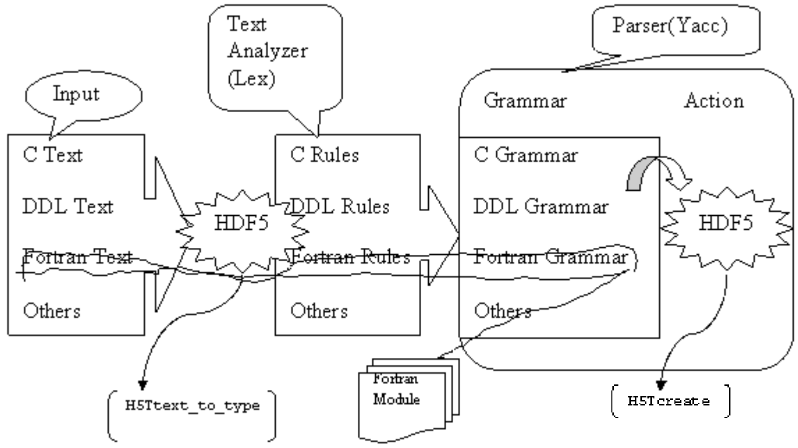

Figure 1. Conversion from text to data type.

From the diagram, we can see there are three major steps, *text input*, *text analysis*, and *parsing*. Input text has to match the format

The parser will take the tokens and symbols generated by the text analyzer and check if they are valid based on the language gram
parser sees valid expression, it will take actions. A token for *“unsigned”* followed by the token for *“int”* is considered as good C
HDF5*(H5Tcreate)* occurs then. A data type object ID is created and returned. It will be illegal to have a token for *“float”* follow

2. Conversion from HDF5 data type to text

This part is relatively simple. Function `H5Ttype_to_text` prints out the text description of an HDF5 data type according to the la
what the `h5dump` does. `h5dump` only prints out the text in DDL format. `H5Ttype_to_text` supports C, DDL, and Fortran.

V. Design Issues to be Considered

1. Modules for languages

The library currently supports C, DLL, and Fortran formats. In the future, we may want to support other languages, like C++. Th
to make future addition simple.

It is possible to provide a tool built with Lex and Yacc. Library designers and users can simply provide a text file consisted of synt
language. By running this tool, rules for Lex and grammar for Yacc can be generated automatically. Then they can be easily plug
could be the second stage of this project if there are enough needs and requirements.

2. Availability of Lex library on systems

There is no need for the Yacc library. The Lex library is optional in order to compile this part of HDF5 library. As long as we defi
overwrite the default Lex function, we do not need to link to any Lex library. The way we use Lex and Yacc tools is similar to the
and Yacc to generate the desired .c and .h files, the code is supposed be portable.

3. Different kinds of Lex and Yacc

For Lex, there are versions of *AT&T* Lex, *GNU* Flex, *POSIX* Lex, etc. For Yacc, there are versions of *AT&T* Yacc, *Berkeley* Yacc
of them may be somehow different from the others. We do not have to address the differences of syntax based on what Lex and Y
GNU Flex and Bison.

4. Error report

A good error report is needed if there are errors in the input text. Both Lex and Yacc have some mechanisms to report errors. We
library error report.

5. Supported data types

For C and DDL, we will support all atomic data types, compound, enumerate, and array types, including their nested cases. We d
opaque data types because C language does not have equivalent data types for them. It will be difficult for the text analyzer and pa
data types.

For Fortran, we will only support four atomic types. But Fortran user can still use DDL description to create data types.

VI. Language Formats

1. C

For text description for C should be the same as C language itself with some minor differences. A text *“unsigned long long”* will
H5T_NATIVE_ULLONG.

For A complete list of data type definition in C language, please refer to the *Appendix B, Syntax of the C language* in the book *C -*
differences we have here is array type. We added array as a data type.

The data types we support are defined in BNF as follows,

```
type-specifier:
      enumeration-type-specifier
      floating-type-specifier
```

```
enumeration-type-definition:
      enum enumeration-tagopt { enumeration-definition-list }

enumeration-type-reference:
      enum enumeration-tag

enumeration-tag:
      identifier

enumeration-definition-list:
      enumeration-constant-definition
      enumeration-definition-list,enumeration-constant-definition

enumeration-constant-definition:
      enumeration-constant
      enumeration-constant = integer-constant

enumeration-constant:
      identifier
```

The floating-point data types are defined as,

```
floating-type-specifier:
      float
      double
      long double
```

The integer data types are defined as,

```
integer-type-specifier:
      signed-type-specifier
      unsigned-type-specifier
      character-type-specifier

signed-type-specifier:
      short or short int or signed short or signed short int
      int or signed int or signed
      long or long int or signed long or signed long int
      long long or long long int or signed long long or signed long long int

unsigned-type-specifier:
      unsigned short intopt
      unsigned intopt
      unsigned long intopt
      unsigned long long intopt

character-type-specifier:
      char
      signed char
      unsigned char
```

The structure data types are defined as,

```
structure-type-specifier:
      structure-type-definition
      structure-type-reference

structure-type-definition:
      struct structure-tagopt { field-list }

structure-type-reference:
      struct structure-tag

structure-tag:
      identifier

field-list:
      component-declaration
      field-list component-declaration

component-declaration:
      type-specifier component-declaration-list ;

component-declaration-list:
```

```
      bit-field:
            declaratoropt : width

      width:
            expression
```

The typedef is defined as,

```
      typedef-name:
            identifier
```

We also support array as a data type in HDF5, which is different from C. Below is the definition of array,

```
      array-declarator:
            type-specifier simple-declaratoropt [ constant-expression ]

      simple-declarator:
            identifier
```

Below is a list of examples of C data types,

Integer types

"char" *"unsigned char"*
"short" *"unsigned short"*
"int " *"unsigned"*
"long;" *"unsigned long"*
"long long" *"unsigned long long"*

Floating-point types

"float" *"double"* *"long double"*

Structures

"struct s {int a; float b;};" *"typedef struct s {int a; float b;} s_t;"*

Arrays

"int [16];"
"typedef struct s {int a; float b;} s_t; s_t [16][32];"

Enumerates

"enum {Bob=0, Elena, Quincey, Frank};"

2. DDL

This format is basically the DDL definition for HDF5. Please look at the last chapter of the *User's Guide for HDF5, DDL for HD*
for this project's concern is as follows,

```
<datatype> ::= <atomic_type> | <compound_type> | <array_type> |
               <variable_length_type>

<atomic_type> ::= <integer>  | <float>  | <time>      | <string> |
                  <bitfield> | <opaque> | <reference> | <enum>

<integer> ::=  H5T_STD_I8BE     | H5T_STD_I8LE      |
               H5T_STD_I16BE    | H5T_STD_I16LE     |
               H5T_STD_I32BE    | H5T_STD_I32LE     |
               H5T_STD_I64BE    | H5T_STD_I64LE     |
               H5T_STD_U8BE     | H5T_STD_U8LE      |
               H5T_STD_U16BE    | H5T_STD_U16LE     |
               H5T_STD_U32BE    | H5T_STD_U32LE     |
               H5T_STD_U64BE    | H5T_STD_U64LE     |
               H5T_NATIVE_CHAR  | H5T_NATIVE_UCHAR  |
               H5T_NATIVE_SHORT | H5T_NATIVE_USHORT |
               H5T_NATIVE_INT   | H5T_NATIVE_UINT   |
               H5T_NATIVE_LONG  | H5T_NATIVE_ULONG  |
               H5T_NATIVE_LLONG | H5T_NATIVE_ULLONG

<float> ::= H5T_IEEE_F32BE   | H5T_IEEE_F32LE     |
            H5T_IEEE_F64BE   | H5T_IEEE_F64LE     |
```

```
<strsize> ::= <int_value>
<strpad> ::= H5T_STR_NULLTERM | H5T_STR_NULLPAD | H5T_STR_SPACEPAD
<cset> ::= H5T_CSET_ASCII
<ctype> ::= H5T_C_S1 | H5T_FORTRAN_S1

<bitfield> ::= TBD

<opaque> ::= H5T_OPAQUE { <identifier> }

<reference> ::= H5T_REFERENCE { <ref_type> }
<ref_type> ::= H5T_STD_REF_OBJECT | H5T_STD_REF_DSETREG

<compound_type> ::= H5T_COMPOUND { <member_type_def>+ }
<member_type_def> ::= <datatype> <field_name> <offset>opt ;
<field_name> ::= <identifier>
<offset> ::= : <int_value>

<variable_length_type> ::= H5T_VLEN { <datatype> }

<array_type> ::= H5T_ARRAY { <dim_sizes> <datatype> }
<dim_sizes> ::= `['<dimsize>`]' | `['<dimsize>`]'<dim_sizes>
<dimsize> ::= <int_value>

<enum> ::= H5T_ENUM { <enum_base_type> <enum_def>+  }
<enum_base_type> ::= <integer>
// Currently enums can only hold integer type data, but they may be //expanded in the future to hold ar
<enum_def> ::= <enum_symbol> <enum_val>;
<enum_symbol> ::= <identifier>
<enum_val> ::= <int_value>
```

A few examples of datatypes in DDL are as follows,

```
"H5T_ENUM { H5T_NATIVE_INT;
      "Bob"       0;
      "Elena"     1;
      "Quincey"   2;
      "Frank"     3;    }"

"H5T_COMPOUND {
      H5T_ARRAY { [4] H5T_STD_I32BE } "int_array";
      H5T_ARRAY { [5][6] H5T_IEEE_F32BE } "float_array"; }"

"H5T_COMPOUND {
      H5T_STD_I16LE     "16_bit"  :  0;
      H5T_IEEE_U32BE    "32_bit"  :  16; }"
```

3. Fortran

To be decided.

VII. Examples

A simple example below shows how to create an array datatype of compound type in C format. This compound data type has two float. The program then converts the HDF5 data type just created into a text description.

```
   hid_t    dtype;
   size_t   tsize;
   unsigned char* text_buf;
            :

   /* Create the data type by C text */
   if((dtype = H5Ttext_to_type("typedef struct foo{
                                     int a;
                                     float b;
                               } foo_t;
                             foo_t [12];"))<0)
            goto error;

   /* Convert the data type back to text */
   If(H5Ttype_to_text(dtype, NULL, H5T_C, &tsize)<0)
            goto error;

   If(tsize>0)
```

Name: H5Ttext_to_type

Signature:

hid_t `H5Ttext_to_type`(*const char** `str`)

Purpose:

Create a HDF5 datatype given a description of data type.

Description:

Given a text description of data type, this function creates an HDF5 datatype. The text description of the data type has to comply with certain language formats. The currently supported languages are C, DDL, and Fortran. An example of C text description is like,

```
“typedef struct foo {
        int a;
        float b;
 } foo_t;
 foo_t [12];”
```

When this C definition of data type is passed in as the `str`, this function will create an HDF5 datatype of 12-element array
compound datatype has a field of integer and a field of float.

Parameters:

*const char** `str`

IN: a character string describing the data type to be created.

Returns:

Returns the datatype ID(non-negative) if successful; otherwise returns a negative value.

Name: H5Ttype_to_text

Signature:

herr_t `H5Ttype_to_text`(*hid_t* `datatype`, *char** `str,` *H5T_lang_t* `lang_type,` *size_t** `len`)

Purpose:

Creates a text description of a datatype.

Description:

Given a datatype ID, this functions creates a text description of this datatype in different format according to the language t
text description will be in C format. If it is `H5T_DDL`, the description will be in HDF5 DDL format. An example in C format will be like,

```
“typedef struct foo {
        int a;
        float b;
 } foo_t;
 foo_t [12];”
```

which is a datatype of 12-element array of a compound datatype. This compound datatype has a field of integer and a field

A preliminary `H5Ttype_to_text` call can be made to find out the size of the buffer needed. This value is returned as `len`.
`len` for a second `H5Ttype_to_text` call, which will retrieve the actual text description for the data type.

If the library finds out `len` is not big enough for the description, it simply returns the size of the buffer needed through `len`
buffer.

Parameters:

hid_t datatype

IN: ID of the datatype to be converted.

*char** `str`

OUT: Buffer for the text description of the data type.

H5T_lang_t `lang_type`

IN: the language used to describe the data type. Currently supported languages are `H5T_C, H5T_DDL, H5T_FORTRAN`. Ot

*size_t** `len`

OUT: the size of buffer needed to store the text description.

Returns:

Returns non-negative if successful; otherwise returns a negative value.

Name: H5Tcmp

Signature:

int `H5Tcmp`(*hid_t* `dtype1, hid_t dtype2`)

Purpose:

	H5T_COMPOUND > H5T_OPAQUE > H5T_BITFIELD > H5T_STRING > H5T_TIME > H5T_FLOAT > H5T_INTEGER
Integers	Big endian > little endian, higher precision > lower precision, greater offset > lesser offset(bit position of the least significant bit), greater least significant padding > lesser least significant padding, greater most significant padding > lesser most significant padding, unsigned > signed.
Floats	Big endian > little endian, higher precision > lower precision, greater offset > lesser offset(bit position of the least significant bit), greater least significant padding > lesser least significant padding, greater most significant padding > lesser most significant padding, greater bit position of sign bit > lesser bit position of sign bit, greater position of least significant bit of exponent > lesser position of least significant bit of exponent, greater exponent size > lesser exponent size, greater exponent bias > less exponent bias, greater mantissa size > lesser mantissa size, most significant bit of mantissa is 1 always > most significant bit of mantissa is implied(normalization), padding set to background value > padding set to 1 > padding set to 0.
Times	Big endian > little endian, higher precision > lower precision, greater offset > lesser offset(bit position of the least significant bit), greater least significant padding > lesser least significant padding, greater most significant padding > lesser most significant padding.
Strings	Big endian > little endian, higher precision > lower precision, greater offset > lesser offset(bit position of the least significant bit), greater least significant padding > lesser least significant padding, greater most significant padding > lesser most significant padding, padding with space for extra bytes > padding with nulls > null-terminated.
Bit fields	Big endian > little endian, higher precision > lower precision, greater offset > lesser offset(bit position of the least significant bit), greater least significant padding > lesser least significant padding, greater most significant padding > lesser most significant padding.
Compounds	More members > less members, greater member name > lesser member name(similar to string comparison), greater member offset > lesser member offset, greater member size > lesser member size.
References	Big endian > little endian, higher precision > lower precision, greater offset > lesser offset(bit position of the least significant bit), greater least significant padding > lesser least significant padding, greater most significant padding > lesser most significant padding, internal reference > dataset region reference > object reference, data located on disk > data located in memory(for object reference),
Enumerates	Greater parent > lesser parent, more members > less members, greater member name > lesser member name.
Variable-lengths	String > sequence, data located on disk > data located in memory, greater file object address > lesser file object address(if they are in different files).
Opaques	H5T_ARRAY > H5T_VLEN > H5T_ENUM > H5T_REFERENCE > H5T_COMPOUND > H5T_OPAQUE > H5T_BITFIELD > H5T_STRING > H5T_TIME > H5T_FLOAT > H5T_INTEGER, greater tag > lesser tag(similar to string comparison).
Arrays	More dimensions > less dimensions, bigger dimensions > smaller dimensions, greater parent > lesser parent.

This function provides the convenience of sorting data types although some of the comparisons are arbitrary.

Parameters:

hid_t `dtype1`

IN: ID of the first datatype to be compared.

hid_t `dtype2`

IN: ID of the second data type to be compared.

Returns:

Returns positive value if first data type is greater; negative value if second data type is greater; 0(zero) if they are equal.