The Pleodata Language and Representation

Alexander Gray Benjamin Bornstein Eric Mjolsness

December 21, 1999

Last updated: 12/21/99 BB

1 Introduction

Pleodata (Pleomorphic Data) is a markup language developed and used by the MLS group. It is primarily used to represent data. Because it is defined very generally, however, it can be used to represent almost anything. Other current and planned uses of Pleodata include representation of models of data, abstract algorithms, and software. It is related to existing languages such as XML, though it has different emphases.

1.1 Pleodata Files

To understand Pleodata, think of a text file containing data. Here's an example of such a file:

```
< mineral &Mineral1
  otherinfo = "Test Line"
  spectra = &Spectrum1
>
< spectrum &Spectrum1
  spectraname = "VISIBLE"
  numpoints = 5
  datamatrix = ( 0.0  1.0
                 1.0  0.5
                 2.0  0.2
                 3.0  0.1
                 4.0  0.05 ) >
>
```

This data file contains textual representations of two Pleodata objects, a mineral and its spectrum. These data can be said to be *self-describing*, in the sense that the numbers and strings that form the parts of a mineral datum are annotated with the additional information indicating that structure. It is a departure from the usual table-formatted data found in relational databases, in which the annotations which describe the nature of each column and row in the table are stored separately from the data themselves.

Note that the mineral object has a name, '&Mineral1', which we call the *object ID*. An object's name can be used to have one object point to another, or refer to another without defining it on the spot. This allows the Pleodata data file to describe an interlinked collection of data, thereby conveying additional information about how they are related.

An alternate Pleodata form for the same data is:

```
< mineral &Mineral1
  otherinfo = "Test Line"
  spectra =
```

```
  < spectrum
    spectraname = "VISIBLE"
    numpoints = 5
    datamatrix = ( 0.0  1.0
                   1.0  0.5
                   2.0  0.2
                   3.0  0.1
                   4.0  0.05 ) >
>
```

Here we've defined the spectrum in place, which might be more convenient in some situations. We also decided not to name the spectrum object, since we don't anticipate any need to reference it in the future.

1.2 Pleodata Internal Structures

The name *'Pleodata'* refers to both a text representation (*Pleodata files*) and an internal memory representation (*Pleodata objects* or *structures*). A program which is to operate upon the data encoded in a Pleodata file must have an internal representation of some sort, such as Matlab's matrices or a C program's data structures. This is usually very specific to the type of data expected. The author of a program that wishes to read in Pleodata files faces the task of translating the Pleodata files into those *application-specific* internal structures.

To make this easier, there is an abstract Pleodata internal representation which can be used as an intermediary: First a Pleodata file is parsed into this internal representation, then the program may manipulate the data as desired, e.g. translate each Pleodata internal object into an application-specific object.

A Pleodata file is a linked list of object *instances*. Each instance contains a linked list of *arguments* or fields. Finally, each argument contains a linked list of *values*. The simple file

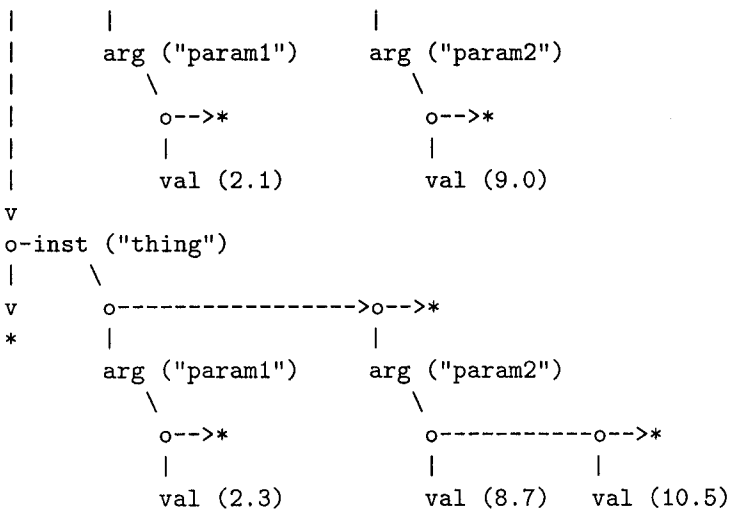
```
< thing
  param1 = 2.1
  param2 = 9.0
>
< thing
  param1 = 2.3
  param2 = (8.7 10.5)
>
```

can be shown schematically as:

```
inst ("thing")
|   \
|    arg ("param1") --> arg ("param2")
|       \                  \
|        val (2.1)          val (9.0)
v
inst ("thing")
    \
     arg ("param1") --> arg ("param2")
        \                  \
         val (2.3)          val (8.7) --> val (10.5)
```

Showing the additional detail of the nodes of the various linked lists, we have:

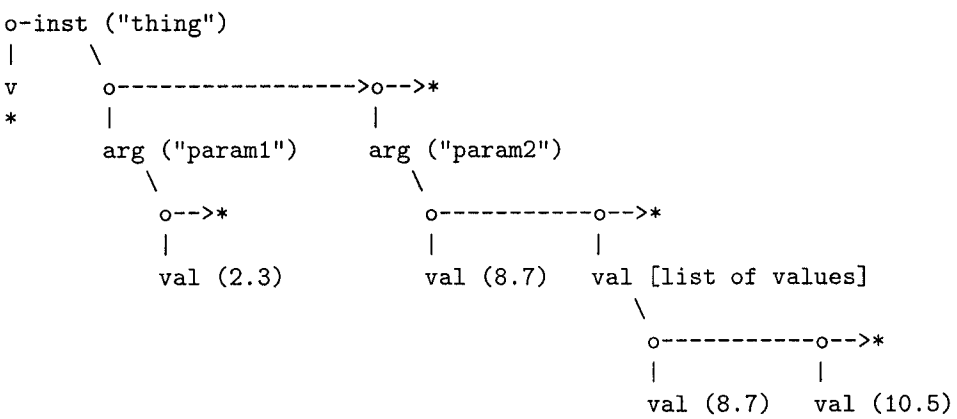
```
o-inst ("thing")
|     \
|      o---------------->o-->*
```



Pleodata allows one other basic structural capability, the idea of nested values. The file

```
< thing
  param1 = 2.3
  param2 = (5 (8.7 10.5))
>
```

would be represented internally as



2 Goals and Principles

The main goals and principles of Pleodata are:

2.1 Representational Generality

It should be clear that this type of representation is quite general and able to describe many diverse types of data, including things that might not normally be thought of as 'data'. Other current and planned uses of Pleodata include representation of models of data, abstract algorithms, and software.

2.2 Data Format Interoperability

The representational generality of Pleodata makes it a natural choice for a common data format, or *lingua franca*, for interoperation of different data-handling programs and software environments. The simplest method for achieving such interoperation is to use Pleodata as the intermediary format from and to which all formats of interested can be translated.

2.3 Object-Oriented Tools

The representational philosophy of Pleodata is exactly that of object-oriented databases and object-oriented programming languages. This should allow smoother conceptual and practical usage of these tools with data of interest.

3 Software and Examples

Pleodata *parsers* take Pleodata text files (usually denoted by the .pleo suffix) and create corresponding internal memory structures. Pleodata *generators* take Pleodata memory structures and create corresponding Pleodata text files. Pleodata *editors* allow input through a GUI for creation of internal memory structures and/or text files.

Here is the set of existing Pleodata software and examples:

- **C Pleodata Parser Example**

  A pleo parser which reads in a Pleodata text file and creates corresponding Pleodata internal structures in the C language can be found in:

  `C/txt2pleo`

  That program prints new Pleodata text files, corresponding to its internal structures, demonstrating the ability to go back and forth between text and memory Pleodata representations. It is the most elementary example of using Pleodata. Note that every application using Pleodata internally must call the Pleodata parsing routines; that's why this example exists, to demonstrate how to use the Pleodata API. Includes sample input and output.

- **C Pleodata Library**

  The C Parser is based on the C library

  `C/pleolib`

  which contains all functions for parsing from Pleodata files, as well as manipulating, and printing Pleodata structures.

- **C Pleodata-to-Minerals Translator Example**

  This is a more realistic example, using Minerals (the data type for an actual geological application) to show how applications with specific data formats can use the Pleodata represenatations. It can be found in:

  `C/pleo2minerals`

  It shows how to read in a Pleodata text file, creating Pleodata internal structures, then creating corresponding Mineral structures and printing out the data in a Mineral text format. Includes sample input and output.

- **C Pleodata-based Application Example**

  This example is more realistic in the sense of a complex data analysis computation. It reads data in Pleodata format, translates the data into its own internal structures, and performs statistical mechanics-based clustering on the data. It can be found in:

  `C/sclust`

  Includes sample input and output.

- **C++ Pleodata Parser Example**

  The C++ analog to the C Pleodata Parser can be found in:

  `C++/txt2pleocc`

  It is meant to do exactly what the C Pleodata Parser does, except that its internal structures are actually C++ objects. The Pleodata objects are defined in part using the Standard Template Library, as implemented by ObjectSpace's Systems¡Toolkit¿. Includes sample input and output.

- **C++ Pleodata Library**

  The C++ Parser is based on the C++ library

  `C/pleocclib`

  This library is meant to completely mirror the C Pleodata Library.

- **C++ Pleodata-to-Minerals Translator Example** This is the C++ analog of the C Pleodata-to-Minerals Translator Example, contained in:

  `C++/pleocc2minerals`

  Includes sample input and output.

- **Perl Pleodata Parser Example**

  The Perl parser can be found in:

  `perl`

  Includes sample input and output, including description of how to call it from C.

- **Java Pleodata Parser Example and Editor**

  The Java parser is combined with an interactive editor, which allows the user to change the contents of a Pleodata file through a graphical user interface. The Java parser and editor can be found in:

  `java/PleoEdit`

  Includes sample input and output.

- **XML-to-Pleodata Parser Example**

  This is an example of reading in XML text files and creating corresponding Pleodata internal structures. It can be found in:

  `C/xml2pleo`

  Includes sample input and output. This shows an example of a DTD designed for an application-specific subset of XML. The format for a document similar to a DTD, except describing an application-specific subset of Pleodata has been defined. Software for converting between the XML-DTD and the proposed 'Pleodata-DTD' is under development.

4 Language Spec

It is important that all implementations of Pleodata parsers and generators follow the following specification of the Pleodata language so that consistency is maintained. A language specification for Pleodata is defined by the following grammar:

```
INSTANCES --> INSTANCES INSTANCE |
              INSTANCES COMMENT
              INSTANCE |
INSTANCE  --> < TYPENAME OBJECTID ARGUMENTS > |
              < TYPENAME OBJECTID > |
              < pleo OBJECTID ARGUMENTS >
TYPENAME  --> NAMESTRING
OBJECTID  --> OIDSTRING |
              null
ARGUMENTS --> ARGUMENTS ARGUMENT |
              ARGUMENT
ARGUMENT  --> ARGNAME = VALUE
ARGNAME   --> NAMESTRING
VALUE     --> OIDSTRING |
              STRINGVAR |
              INTVAR |
              FLOATVAR |
              DOUBLEVAR |
              INSTANCE |
              ( VALUES )
VALUES    --> VALUES VALUE |
              VALUE
COMMENT   --> COMMENTSTRING
```

The 'pleo' keyword may consist of any mixture of upper and lower case. OIDSTRING is (almost) any string beginning with '&'. NAMESTRING is (almost) any string not beginning with a number and not beginning with '&'. STRINGVAR is (almost) anything between any pair of quotations (', ', or "). INTVAR is any integer. FLOATVAR is any floating point number. DOUBLEVAR is any floating point number immediately followed by 'd' or 'D'. A COMMENT is denoted by '#' at the beginning of a line; it is ignored by the parser.

The first line of a Pleodata file can have the special form

```
< pleo syntax-style = STRINGVAR version = FLOATVAR homeinst = STRINGVAR >
```

indicating the Pleodata syntax-style and version, and optionally the home institution where the Pleodata was generated.

More exact definitions are given in terms of the precise regular expressions used in the scanning phase of the C Pleodata parser, found in `pleo_parse_tuple_v2.l` in `/proj/code/c/pleo`.

5 Internal Representation

The set of data structures used for all of the Pleodata parsers to date is described abstractly in the introduction. Its actual implementation is captured by the following excerpt from `pleo_util.h`:

```
/* instance */
struct PLEO_instance_struct {
  char                      *type_name; /* string name of object type */
  char                      *object_id; /* string name of this instance */
  struct PLEO_list_struct   *arguments; /* list of arguments, or arg structs */
};
```

```
/* arg */
struct PLEO_arg_struct {
  char                    *arg_name;      /* string name of argument type */
  struct PLEO_list_struct *values;        /* list of values, or val structures */
};

/* val */
struct PLEO_val_struct {
  int                          type;          /* code for value's type */
  struct PLEO_instance_struct *instance;      /* setting if type is an instance
                                                 or an oid */
  char                         *oid_val;      /* setting if type is an oid */
  char                         *string_val;   /* setting if type is string */
  union                        {
  int                            int_val;     /* setting if type is int */
  float                          float_val;   /* setting if type is float */
  double                         double_val;  /* setting if type is double */
                               } numeric;
  struct PLEO_list_struct      *array_val;    /* setting if type is array */
};
```

6 Programming API

Manipulation of internal Pleodata structures is accomplished though the Pleodata API (application programming interface):

```
Pleo Utility Functions
(functions that assist in reading/creating/writing/searching pleo C structures)

Pleo Parsing:
------------

PLEO_PARSE_INIT
Set up variables for parsing.
int PLEO_parse_init( int the_syntax_style, FILE* in_fp )

PLEO_PARSE
Parse a Pleo text file, by calling the appropriate flex/bison-generated parse
function.
int PLEO_parse( )

PLEO_CONNECT_INSTANCE_PTRS
Traverse a list of PLEO_instances, connecting OID (object ID) pointers to
PLEO_instances within the list.
Note that this routine expects a flat list of all instances, not one in which
some instances are not in the main backbone of the list.
PLEO_list *PLEO_connect_instance_ptrs( PLEO_list *the_instance_list )

Constructors:
------------

PLEO_START_LIST
Make a new list, starting with a specified element as the head.  Returns the
```

```
list.
PLEO_list *PLEO_start_list ( void *the_element, int list_type )

PLEO_MAKE_LIST
Allocate a new, empty PLEO_list structure.
PLEO_list *PLEO_make_list ( )

PLEO_MAKE_INSTANCE
Allocate a new, empty PLEO_instance structure.
PLEO_instance *PLEO_make_instance ( )

PLEO_MAKE_ARG
Allocate a new, empty PLEO_arg structure.
PLEO_arg *PLEO_make_arg ( )

PLEO_MAKE_VAL
Allocate a new, empty PLEO_val structure.
PLEO_val *PLEO_make_val ( )

PLEO_APPEND_TO_LIST
Add an element to a list of objects.  Returns the list.
PLEO_list *PLEO_append_to_list ( void *the_element, PLEO_list *the_list )

(We need more functions here to make it simpler & more direct to construct
 pleo structures from application-specific structures.  This is currently
 under development.)

Pleo Output:
------------

PLEO_WRITE_INSTANCE
Write the contents of a PLEO_instance structure to a file.  Does not expand
OID's.
int PLEO_write_instance ( char *file_name, PLEO_instance *the_instance,
                          char *mode )

PLEO_WRITE_LIST
Write a list to a file.  Does not expand OID's.
int PLEO_write_list( char *file_name, PLEO_list *the_list, char *mode )

PLEO_PRINT_INSTANCE
Print the contents of a PLEO_instance structure to a stream.
int PLEO_print_instance ( FILE *stream, PLEO_instance *the_instance )

PLEO_PRINT_INSTANCE_EXPAND
Print the contents of a PLEO_instance structure to a stream, expanding OID's
into their full instance form.
int PLEO_print_instance_expand ( FILE *stream, PLEO_instance *the_instance )

PLEO_PRINT_LIST
Print a list to a stream.
int PLEO_print_list ( FILE *stream, PLEO_list *the_list )

PLEO_PRINT_LIST_EXPAND
```

```
Print a list to a stream, expanding OID's into their full instance form.
int PLEO_print_list_expand ( FILE *stream, PLEO_list *the_list )

Pleo Search Functions:
----------------------

PLEO_FIND_INSTANCE_NAMED
Return the PLEO_instance in a list of instances that matches the specified
object id.
PLEO_instance *PLEO_find_instance_named( PLEO_list *the_instance_list,
                                         char* object_id )

PLEO_FIND_ARG_NAMED
Return the PLEO_arg in a list of args that matches the specified argument name.
PLEO_arg *PLEO_find_arg_named( PLEO_list *the_arg_list, char* arg_name )

PLEO_RFIND_INSTANCE_NAMED
Return the PLEO_instance contained in a list of instances or their children
(i.e., recursive find) that matches the specified object id.
PLEO_instance *PLEO_rfind_instance_named( PLEO_list *the_instance_list,
                                          char* object_id )

PLEO_RFIND_ARG_NAMED
Return the PLEO_arg contained in a list of args or their children (i.e.,
recursive find) that matches the specified argument name.
The instance list defines the scope of unconnected OID searches.  (Note:
OID searches are not needed for OID types that have "connected" instance
pointers, e.g. an instance produced via parsing a pleo text file.)  If
the instance list is NULL the function will not try to recurse on unconnected
OID type values.
PLEO_arg *PLEO_rfind_arg_named( PLEO_list *the_arg_list, char* arg_name,
                                PLEO_list *the_instance_list  )

PLEO_RFIND_INSTANCE_TYPE_IN_LIST
Return the PLEO_instance contained in a list of instances or their children
(i.e., recursive find) that matches the specified instance type name.
PLEO_instance *PLEO_rfind_instance_type_in_list( PLEO_list *the_instance_list,
                                                 char* type_name )

PLEO_RFIND_INSTANCE_TYPE
Return the first PLEO_instance contained in a list of instances or their
children instances (recursive find) that matches the specified instance type
name.
The instance list defines the scope of unconnected OID searches.  (Note:
OID searches are not needed for OID types that have "connected" instance
pointers, e.g. an instance produced via parsing a pleo text file.)  If
the instance list is NULL the function will not try to recurse on unconnected
OID type values.
PLEO_instance *PLEO_rfind_instance_type( PLEO_instance *the_instance,
                                         char* type_name,
                                         PLEO_list *the_instance_list )

PLEO_COPY_ARG_VAL
Locates pleo argument (field) by name in the given pleo instance, and copies
```

```
the value into destination after casting.  The value and destination types
must match.  Returns ERROR if named argument is not found, or if the argument
and destination value types do not match.
int PLEO_copy_arg_val ( char *arg_name, int dest_val_type,
                        PLEO_instance *the_instance, void *dest )

PLEO_EXTRACT_TWOD_ARRAY
Recursively searches instance for an instance of type twod_array, then
copies the contents of the pleo into a newly allocated 2-D data_array
of floats.  The instance "twod_array" should have "rows", "columns"
and "values" arguments (fields).
int PLEO_extract_twod_array (PLEO_instance *the_instance,
                             float ***data_array, int *rows, int *cols)

PLEO_OID_IS_CONNECTED
Return 1 of this pleo value is an OID and the instance pointer has been
connected (resolved), and 0 otherwise.  This will generally return true
for instances in a list that was run through PLEO_connect_instance_ptrs().
int PLEO_oid_is_connected (PLEO_val *the_val)
```

The API is currently undergoing a design iteration and some changes are very likely in the near term.

7 Status

Pleodata is still an evolving entity. As working data analysis systems are being built using Pleodata, its capabilities are likely to be extended and more software to manipulate Pleodata and use it as a translation base is likely to be built. As reliance on Pleodata increases, however, both by system builders within the MLS group and by collaborators, radical backward-incompatible changes shouldn't be expected. This document is intended to describe the very latest state of Pleodata.

Pleodata's user base currently includes:

- GRN Computational Biology Project (MLS)
- Chris Hart (Caltech Biology Dept.)
- Wolfgang Fink (Caltech Physics Dept.)

8 Misc. Notes

1. Pleodata used to be called 'Glu'. This is now outdated.

9 People and History

Pleodata began with preliminary design work by Eric Mjolsness. Alex Gray wrote the first Pleodata library and parsers based on it, in C and C++ versions, and examples of their use and is the original author of this document. Eric wrote the first Perl parser. Ben Bornstein wrote the first Java parser and editor. Tobias Mann, Becky Castano, and Joe Roden helped refine and make Pleodata a reality by using Pleodata for real data analysis tasks. Vlad Gluzman and Alex extended Pleodata to allow nested lists of values. Vlad wrote an XML-to-Pleodata parser. Based on suggestions by Mike Turmon and others, Joe extended the Pleodata manipulation API.

10 Contact

Eric Mjolsness **emj@aig.jpl.nasa.gov**