Polyglot Input/Output Library

Documentation

Version 1.0

François Clément Pierre Weis

2014-06-11

1 Outline

Pio is a safe and robust polyglot communication library.

2 What is Pio?

The Pio system is a set of communication libraries to safely exchange values between programs written in various languages. For the time being Pio offers a library for OCaml, C, C++, and Fortran.

3 The Pio library specification

We describe here the overall definition and structure of the Application Programmer Interface, or API, of the Pio library, Version 1.0.

3.1 The concept of communication

The Pio library defines and implements values, called *communications*, that can be safely exchanged between programs written in different languages.

Any communication that satisfies the following specification (both the syntactic specification and the semantic restrictions) must be accepted without errors (except for hardware problems on the wires). The semantics of the transmitted values must not be modified by the transmission process: ping-pong communications should be a no-op (i.e. the value read back is equal to the value sent).

We exchange communications as described below. Communication basic blocks are named *typed values*; a typed value is a value tagged with its type.

3.2 The description of communications

A communication can be:

- a `Phrase` communication, *i.e.* a sequence of typed values;
- a `Task` communication, *i.e.* a function name and its arguments;

- a `Result` communication, *i.e.* a sequence of typed values;
- an `Error` communication, *i.e.* an exception value;
- a `Service` communication, *i.e.* a meta information about the communication.

A `Task` communication is supposed to specify a function to be applied to its arguments by the receiver; the function is given by its name (a string) and the arguments (a sequence of typed values).

The intended meaning of `Result` and `Error` communications is to tag the effect of a previous `Task` communication: if the function application succeeds, the result is sent back as a `Result` communication, otherwise an exception value is sent back using an `Error` communication.

A `Service` communication is supposed to carry some information about the communication process: acknowledgement of communication, request for beginning/end of communication, request for initialization/finalization of the reader program, and the library error communication about bogus communication; each service message may have some associated typed values.

If any data in the communication is not compliant to the grammar and to the various constraints expressed in the comments, then the entire communication is invalid. In that case, the library should not produce any value (even possibly truncated ones) but must return `Service` "WrongCommunication" to the calling program.

The library communicates binary values encoded in so-called *little* endian. Both ends of the communication have not to worry about endianness: the library takes care of the endianness when appropriate.

The OCaml implementation of Pio uses buffers. The user must provide appropriate i/o buffers (see the Buffer and Scanf.Scanning modules).

4 The grammar of communications

4.1 Communication description

We describe here the precise string of characters that defines a valid communication.

Communication ::=
```
  '(' '\n'            (* Beginning of communication marker. *)
  Phrase | Task | Result | Error | Service
  ')' '\n' '\n'       (* End of communication marker. *)
```

Phrase ::=
```
  "%p" Values_number n '\n'      (* Phrase tag, and number of values n. *)
  Typed_value*                    (* n typed values. *)
```

Task ::=
```
  "%t" Values_number n ' ' Task_name '\n'
                      (* Task tag, number of arguments n, and task/function name. *
  Typed_value*        (* n typed values. *)
```

```
Result ::=
  "%r" Values_number n '\n'        (* Result tag, and number of results n. *)
  Typed_value*                      (* n typed values. *)

Error ::=
  "%e" Values_number n '\n'        (* Error tag, and number of values n. *)
  Typed_value*                      (* n typed values. *)

Service ::= "%s" ' ' Service_name '\n'        (* Service tag, and service name. *)

Task_name ::= String

Values_number ::= ' ' Size ' '        (* A size surrounded with spaces. *)

Size ::= '<' Natural '>'              (* A delimited natural. *)

Natural ::= Decimal_digit+            (* Usual 10-basis representation. *)

Decimal_digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

4.2 Service description

```
Service_name ::=          (* Intended meaning from the client's (the program that re
                             the communication) point of view. *)
  | "Ok"                  (* Acknowledge service: the preceding communication was un
  | "Ko"                  (* Negative acknowledge service: the preceding communicati
                             not understood. *)
  | "Allo"                (* Communication requested. *)
  | "Bye"                 (* End of communication requested. *)
  | "Start"               (* Initialization of program requested. *)
  | "Stop"                (* Finalization of program requested. *)
```

5 Typed value description

For the time being, typed values are restricted to:

- one boolean ("true" or "false");
- one string (a sequence of bytes);
- one integer;
- one float;
- one boolean couple, or triple;
- one integer couple, or triple;
- one float couple, or triple;

- one vector of such;
- one integer matrix;
- one float matrix.

To prevent erroneous interpretation, a typed value contains its type.

In addition, to enforce integrity of the communication, all typed values are delimited between two balanced markers.

```
Typed_value ::=
  "begin" '\n'                          (* Beginning of typed value marker. *)
  ["let" Ident_lexem " =" '\n']         (* Optional name for the typed value. *)
  Value_type '\n'                       (* The type of the following value. *)
  Value                                 (* A value of the preceding type. *)
  "end" '\n' '\n'                       (* End of typed value marker. *)
```

5.1 Type of value description

The types of values is a simplified form of a real language type algebra, although types for Pio encapsulate some size information for arrays (*i.e.* integer values).

```
Value_type ::= Scalar_type | Array_type

Array_type ::= Vector_type | Multi_d_type

Vector_type ::= "[1" Scalar_type "1]"

Multi_d_type ::= '[' Natural p Scalar_type Natural p ']'

Scalar_type ::= Simple_type | Tuple_type

Tuple_type ::=
  | '(' Simple_type ", " Simple_type ')'                    (* Couple type. *
  | '(' Simple_type ", " Simple_type ", " Simple_type ')'   (* Triple type. *

Simple_type ::= Bool_type | String_type | Number_type

Bool_type ::=
  | "%B"
    (* Boolean type,
       declare this value as a C int on any architecture,
       this value is an OCaml bool on any platform. *)

String_type ::=
  | "%S"
    (* String type,
       declare this value as a C char* on any architecture,
       this value is an OCaml string on any platform. *)
```

```
Number_type ::=
  | "%i"
    (* Integer type,
       declare this value as a C int on any architecture,
       this value is an OCaml int on any platform. *)
  | "%ni"
    (* Native integer with exactly one word of information:
       this value is a C pointer on any architecture, so declare this value as a
long
       on any architecture, (including LLP64 for which it could have been a long
long),
       this value is an OCaml Nativeint.t on any platform.
       Note: in case of communication from a 32-bit to a 64-bit architectures, th
       must return the corresponding 64-bit long value; in the converse situation
       can be truncated without loss of information, it must be truncated, otherw
       must fail. *)
  | "%li"
    (* Integer with exactly 32 bits of information,
       declare this value as a C int on any architecture (as a consequence, on an
       architecture the C programmer should not use type _int32),
       this value is an OCaml Int32.t on any platform. *)
  | "%Li"
    (* Integer with exactly 64 bits of information,
       declare this value as a C long long on a 32-bit architecture, and as a C
long
       on a 64-bit architecture (or a long long on LLP64),
       this value is an OCaml Int64.t on any platform. *)
  | "%f"
    (* Float type in usual human readable representation,
       declare this value as a C double on any architecture,
       this value is an OCaml float on any platform. *)
  | "%bf"
    (* Float type in binary representation,
       declare this value as a C double on any architecture,
       this value is an OCaml float on any platform. *)
```

5.2 Value description

A value is either a scalar, or an array. A scalar is either a simple lexem, or a tuple lexem with the usual syntax.

```
Value ::= Scalar_item | Array_item

Scalar_item ::= Scalar ';' '\n'
Array_item ::=  Array ';' '\n'
```

```
Scalar ::= Simple_lexem | Tuple_lexem

Tuple_lexem ::=
  | '(' Simple_lexem ", " Simple_lexem ')'
  | '(' Simple_lexem ", " Simple_lexem ", " Simple_lexem ')'
```

5.2.1 Simple lexems

A simple lexem is either a Pio value ident lexem (*i.e.* a simple form of identifier, which is also an OCaml value ident), or a simple OCaml lexem (*i.e.* a bool, a string, or a number lexem). String lexems also incorporate their size.

```
Simple_lexem ::=
  | Ident_lexem                                    (* A reference to an already named t
  | Bool_lexem | String_lexem | Number_lexem

Ident_lexem ::= [_a-z][0-9_a-z]*                   (* A Pio value ident. *)

Bool_lexem ::= "true" | "false"                    (* A valid OCaml boolean. *)

String_lexem ::= Size '\n' String
    (* A delimited string with its size.
       Quoted characters are counted as only one character in the size. *)

String ::= '"' Char* '"'                           (* A valid OCaml delimited string. *

Char ::= Escaped_char | Printable_char             (* A valid Caml character. *)

Escaped_char ::=
  | '\' Decimal_digit Decimal_digit Decimal_digit
    (* The decimal value of the ASCII code of the character (between 0 and 255).
  | "\b" | "\n" | "\r" | "\t"
    (* The backspace (BS, ASCII 8), linefeed (LF, ASCII 10), carriage return (CR,
       and horizontal tabulation (TAB, ASCII 9) characters. *)
  | "\\" | "\"" | "\'" | "\ "
    (* The baskslash (ASCII 92), double quote (ASCII 34), single quote (ASCII 39)
       and space (SPC, ASCII 32) characters. *)

Printable_char ::= '[32-33]' | '[35-91]' | '[93-126]'
    (* The set of printable characters (letters, digits, punctation characters, a
       More precisely, printable characters are characters whose ASCII code range
       to 126, excluding 34 (double quote), and 92 (backslash).
       Note: printable characters can be escaped (i.e., considered as Escaped_char,
       encoded a 3-decimal-digit ASCII code). *)

Number_lexem ::= Integer | Float | Binary_number
```

```
Integer ::= ⟨a valid OCaml lexem for an integer⟩
Float ::= ⟨a valid OCaml lexem for a floating point number⟩
    (* See the OCaml documentation for those specifications. *)

Binary_number ::= '&' Size sz ⟨sz bytes⟩
    (* The encoding of a floating-point number in binary representation.
       The size is the number of bytes of the binary representation. For instance
       the 64-bit encoding representation of a floating-point number according to
       the IEEE-754 standard. *)
```

5.2.2 Arrays

An array is either a vector, or a multi-dimensional array. Both incorporate their dimension and size(s), and multi-dimensional arrays also incorporate their layout. Rows of a multi-dimensional array are similar to a vector without its size.

```
Array ::= Vector | Multi_d

Vector ::=
  "[1"  '\n'              (* Beginning of vector marker. *)
  Size  '\n'              (* Size of the vector. *)
  Scalar_item*            (* Items of the vector. *)
  "1]"                    (* End of vector marker. *)

Multi_d ::=
  '[' Natural p '\n'      (* Beginning of multi-d array of dimension p marker, w
1. *)
  Multi_d_sizes  '\n'     (* The p sizes of the multi-d array. *)
  Multi_d_layout '\n'     (* The layout of the multi-d array. *)
  Row_item*               (* 1D-rows of the multi-d array.
                             When Π_{i=1}^{p} sz_i = 0, there is no row here. Otherwise, the
                             of rows is Π_{i=2}^{p} sz_{s(i)} where (sz_1, sz_2, ..., sz_p) is the size o
                             multi-d array and s is the permutation defining the
  Natural p ']'           (* End of multi-d array of dimension p marker. *)

Multi_d_sizes ::= '<' Natural_list2 '>'

Multi_d_layout ::=
  | Natural_list2         (* Permutation of the range [0..p-1] expressing the lay
  | 'C'                   (* C-like layout, or line-major layout, corresponding
                             the identity permutation. *)
  | 'F'                   (* Fortran-like layout, or column-major layout, correspo
                             the reverse order permutation. *)

Natural_list2 ::=                 (* A comma-separated list of at least 2 inte
  | Natural ", " Natural          (* the two last integers *)
  | Natural ", " Natural_list2    (* An integer followed by the rest of the li
```

```
Row_item ::= Row ';' '\n'

Row ::=
   "[|" '\n'                    (* Beginning of row marker. *)
   Scalar_item*                 (* Items of the row. *)
   "|]"                         (* End of row marker. *)
```

5.2.3 An example: matrix specification

Rectangular arrays, or matrices, are a particular case of multi-dimensional arrays, more precisely 2D-arrays. The concrete syntax for matrices will thus be the following.

```
Matrix ::=
   "[2" '\n'                                (* Beginning of matrix marker. *)
   '<' Natural ", " Natural '>' '\n'        (* Size of the matrix, lines then colu
   ('C' | 'F') '\n'                         (* Layout of the matrix. *)
   Row_item*                                (* Rows (lines or columns) of the matr
   "2]"                                     (* End of matrix marker. *)
```