A LaTeX to OpenMath Phrasebook

David Malone <dwmalone@maths.tcd.ie>

December 6, 2000

This LaTeX to OpenMath phrasebook is in essentially two parts. The first part (`om2pl`) is a C program which uses the INRIA C library to read a piece of OpenMath and produce Perl code which represents the OpenMath. The second part (`pl2la`) is a Perl program which reads this Perl code and outputs LaTeX.

A short Perl program (`om2la`) which runs `om2pl` and `pl2la` is sequence is also provided.

While designing the program I felt it was important that it should be possible to customise the program relatively easily because:

- is likely that people will be using Content Dictionaries which the program doesn't provide for "natively",
- the formatting of Mathematics is often a personal issue, and people may want to modify the default behavior for standard Content Dictionaries.

This suggested to me that the converter should be written in a language such as Perl, which is widely available, allows easy modification of existing code and is suited to text processing.

Perl5 also allows the representation of recursive data structures using references and anonymous data. This allows for an encoding of OpenMath in Perl which is readable, similar in spirit to the XML encoding and (most importantly) parsable as a valid piece of Perl. This means that once you have a piece of OpenMath in its 'Perl encoding' you don't need to write a parser to use it from Perl.

In this encoding each node in the tree is represented by a Perl hash[1]. Each node has a key `type` which tells you what sort of node it is (eg. binding, application, integer), and several other keys corresponding to information corresponding to that type (eg. name and CD for a OpenMath Symbol). Each Hash is delimited by { and }, and lists of things are delimited by [ ]. Strings are usually single quoted, but may be quoted in other ways if necessary. Figure shows the types of nodes and the keys and values they include.

`Om2pl` takes OpenMath and outputs a Perl subroutine which returns a list of references to OpenMath objects in this encoding. Figure 2 shows a short piece of XML and Figure 3 shows the resulting Perl Encoding.

`Pl2la` is driven by generic code which should allow other phrasebooks to be written and plugged in. It is based around the idea that `format_type` will be called for each node in the tree, and that the driver code[2] will figure out what formatting code should be run. The idea is that simple types are always formatted by the generic type and:

- symbols know how to format themselves,
- applications expect their head to know how to do the formatting,
- bindings expect their binder to know how to do the formatting,
- attributions expect the attributes to know how to do the formatting,
- errors expect the error symbol to know how to do the formatting.

Type	Key	Value
integer	value	value as numeric string
symbol	name	name of symbol as string
	cd	CD name as string
variable	name	variable's name as string
float	value	value as numeric string
string	value	value as string
bytearray	value	value as string
application	head	reference to object you are applying
	args	reference to list of arguments
binding	binder	reference to object doing the binding
	vars	reference to list of possibly attributed variables
	body	reference to object in which variables are bound
attribution	object	reference to object to which attributes apply
	attrs	reference to list of atps
error	symbol	reference to symbol representing error
	args	reference to list of arguments to symbol
atp	attribute	reference to symbol for attribute type
	value	value of attribute

Figure 1: Keys, Values and Type names for Perl encoding

```
<OMOBJ>
  <OMA>
    <OMS name="plus" cd="arith1"/>
    <OMI>6</OMI>
    <OMV name="x"/>
  </OMA>
</OMOBJ>
```

Figure 2: An example piece of OpenMath

```
{
  'type'  => 'application',
  'head'  => {
    'type'  => 'symbol',
    'name'  => 'plus',
    'cd'    => 'arith1',
  },
  'args'  => [{
    'type'  => 'integer',
    'value' => '6',
  },
  {
    'type'  => 'variable',
    'name'  => 'x',
  },
  ],
}
```

Figure 3: The Perl encoding of the sample OpenMath.

Type	Calls function refered to by
integer	`$generic_format_type{'integer'}`
symbol	`$cdformatter{$cd}->{$name}->{'symbol'}` or `$cdformatter{$cd}->{'generic'}` or `$generic_format_type{'symbol'}`
variable	`$varaibleformatter{$name}->{'variable'}` `$generic_format_type{$name}->{'variable'}`
float	`$generic_format_type{'float'}`
string	`$generic_format_type{'string'}`
bytearray	`$generic_format_type{'bytearray'}`

application	If head is a symbol:
	`$cdformatter{$cd}->{$name}->{'application'}`
	`$cdformatter{$cd}->{$name}->{'generic'}`
	`$generic_format_type{'application'}`
	If head is a variable:
	`$varaibleformatter{$name}->{'application'}`
	`$generic_format_type{'application'}`
	Otherwise:
	`$generic_format_type{'application'}`

binding	Similar to application, using the binder.
attribution	Treated as list of single attributes.
	Similar to application, using attribute in the atp,
error	Similar to application, using the error.

For example, suppose that we are about to format a symbol. It will check if code is available for formatting this specific symbol, and if it is not then it calls code which knows how to format generic symbols. The rules used to decide what code to run are shown in Figure fig:howfmt.

Once a node has been formatted its formatting is stored, incase this node needs to be formatted again. In the case of variables some generic functions are provided to deal with the binding of variables, so that their formatting can be kept consistent across the binding.

Slotting into this generic frame work is a formatter written to output LaTeX[3] This formatter has a basic set of rules for producing LaTeX, which it uses if it cannot find specific formatters for the symbols and variables it encounters. These basic formatting rules are as follows:

- Floats, strings and integers just use their value. Bytearrays are formatted as a string of hex digits.
- Symbols are formatted as their name.
- Variables are formatted as their name, unless this formatting is already in use. In this case we subscript the variable.
- Applications are formatted as head$(a_1, a_2, \ldots)$.
- Bindings are formatted as binder.$v_1, v_2, \ldots \rightarrow$ body, or just as the body.
- Attributions are formatted as a list of properties, or just the object.
- Errors are formatted as an "An error of type error occurred".

When a variable or symbol is encountered the formatter will try to load a Perl file from `tex/-variable.pl` or `tex/cdname.pl`. This code may be as simple or as complicated as necessary. For simple example, the symbol gamma in the nums CD essentially returns `\gamma`.

[1]You can think of a hash as a list of keys and values
[2]The driver code is in `format_type.pl`.
[3]See `tex_format.pl`.

$$6 + x \tag{1}$$

Figure 4: The LaTeX version of the Perl.

A complicated example would be partial differentiation, which tries to extract the list of variables which it is differentiating with respect to, and then find the formatting of each of these variables. Some code which can format functions of n variables, infix operators and binary relations is provided in `tex/-useful.pl`, and many of the CDs use this code.

The formatting information I am using within the LaTeX phrasebook essentially contains 4 pieces of information:

fmt The full LaTeX formatting of the object.

sfmt A simple formatting of the object. Usually this is the same as fmt. In the case of a binding it will usually just be the body of the binding. In the case of an attribution this will be the object without the "x has procerity blah". The formatter usually uses this simple formatting as opposed to the full formatting.

lpren What to use as a left bracket if someone wants to bracket this expression.

rpren What to use as a right bracket if some wants to bracket this expression.

The result of feeding our simple example into this LaTeX phrasebook is shown in Figure 4.

Some futher enhancements to the design could be made, including attaching functions to formatted objects which know how to format that object when it arises in a particular contexts. It should be possible to use the generic framework with relatively little modification to produce a phrasebook for something like Mathematica or Maple.