Melt User Manual

1 Introduction

1.1 What is Melt?

Motivation LaTeX is great to format text. OCaml is great to program. What if you want to program LaTeX documents using the OCaml syntax and type system? Then you can use Melt. Let's see how we can produce the following:

Let $X \subset \{1, 2, 3\}$.

With LaTeX, this is easy:

```
Let $X \subset \{1, 2, 3\}$.
```

What does it mean to program this in OCaml? Let's assume that you have some `Latex` library providing the following functions: `t` which converts a string to LaTeX text, `m` which converts a string to LaTeX math, some basic LaTeX macros and `c` which concatenates a list of LaTeX expressions. Our example may then be encoded like this:

```
c [ t "Let "; m "X"; subset; lbrace; m "1, 2, 3"; rbrace;
    t "." ]
```

This is much more verbose than the original LaTeX code! The Melt pre-processor allows lighter syntax:

```
"Let $X {subset} \{1, 2, 3\}$."
```

Note how close this is to the LaTeX syntax. The main differences are: usage of double quotes to enter *text* mode and usage of braces to enter *code* mode. The code mode, here, is used for command `subset`, which is an OCaml value of LaTeX type.

What is Melt? Melt allows you to write LaTeX documents using OCaml. It is composed of the `Latex` library for OCaml, the Melt preprocessor, the Melt tool, and the `Melt` library.

The `Latex` library provides an OCaml interface to LaTeX. It allows to create LaTeX values in text or math mode and to concatenate them. It defines basic LaTeX commands as OCaml values, such as `subset`.

It is possible to write an entire document using only the `Latex` library, but the syntax of OCaml is awful for this purpose, and the document quickly becomes an awful, unreadable mess. The Melt preprocessor provides a much lighter syntax, allowing the user to easily interleave LaTeX values and OCaml code.

The Melt tool takes a Melt program and compiles it into a Postscript or a PDF document. It runs the preprocessor if needed, compiles the program, executes it to produce a LaTeX program, and then runs LaTeX to produce the final document.

The `Melt` library contains various tools to glue things together. In particular, it contains functions to easily include MLPOST pictures.

1.2 Getting Started

A basic Melt program, `hello.mlt`, looks like this:

```
emit (document "Hello, world!")
```

Let's see how to compile this document, and how does it work.

Compiling Using the Melt Tool. You can compile `hello.mlt` in one command using the Melt tool:

```
melt hello.mlt
```

This produces a file `hello.ps` containing the text "Hello, world!". You can produce a PDF file instead, using the `-pdf` option.

Compiling by Hand. Instead of using the Melt tool, you can apply the preprocessor yourself:

```
meltpp hello.mlt -o hello.ml -open Latex -open Melt
```

This produces a file `hello.ml` which looks like this:

```
open Latex;;
open Melt;;
emit (document (mode T ((text "Hello, world!"))))
```

As you can see, the string `"Hello, world!"` is no longer a string but a value of type `Latex.t`. You can them compile this OCaml program. Once executed, it will produce a file `hello.tex` looking like this:

```
\documentclass{article}
\author{}
\begin{document}
  Hello, world!
\end{document}
```

You can compile this file using `latex` or `pdflatex`.

Understanding the Code. The `emit` function of the `Melt` library takes a value of type `Latex.t`. Values of this type describe LaTeX abstract syntax trees (ASTs), *i.e.*, pieces of LaTeX documents. The function writes this AST into the file `hello.tex`.

What we want to emit is a full LaTeX document. The `document` function of the `Latex` library takes a body, of type `Latex.t`, and returns another value of type `Latex.t`. The returned AST contains the document class, the `document` environment, the author and so on.

To produce the body of the document, we use the Melt preprocessor. In a Melt document, double quotes do not create values of type `string` but values of type `Latex.t`. So, here, `"Hello, world!"` is not a `string` but a piece of LaTeX AST that we can give to the `document` function.

Text and Math Modes. You can insert math formula using the dollar delimiters `$`$\cdots$`$`, as in LaTeX:

```
"Assume $x+y=42$."
```

This will produce:

Assume $x + y = 42$.

The world "Assume", as well as the ending dot, are printed in *text* mode, whereas the math formula is printed in *math* mode. You enter text mode using double quotes `"`$\cdots$`"`, and you enter math mode using dollars `$`$\cdots$`$`. Note that you can enter math mode directly:

```
emit (document $x+y=42$)
```

You can also enter text mode while in math mode:

```
$x+y=42 " but " z+t=69$
```

This will produce:

$x + y = 42$ but $z + t = 69$

These modes can be nested arbitrarily, but do not abuse this feature as your code will become less readable.

Code Mode. The third mode is called *code* mode. It is introduced using round brackets delimiters `{`$\cdots$`}`. It allows to insert an OCaml value of type `Latex.t`. In particular, this is useful to write macros:

```
let p = $3.141592$ in
emit (document "{pi} is {p}.")
```

This will produce:

```
π is 3.141592.
```

`pi` is a value of type `Latex.t` which is defined in the `Latex` module.

In code mode, you can insert OCaml code of any complexity:

```
$1+2+3 = {latex_of_int (1+2+3)}$
```

This will produce:

$1 + 2 + 3 = 6$

The code in code mode is also preprocessed. This means that you can nest code, math and text modes arbitrarily. For example:

```
"This {emph "word"} is emphasized."
```

This will produce:

This *word* is emphasized.

Verbatim Mode. The last mode is called *verbatim* mode. It can be used to insert text that should not be parsed, such as code listing[1]. Here is an example:

```
"Code: <<printf "Toto has $%d to spare." 42>>"
```

This will produce:

Here is some code: printf "Toto has $%d to spare." 42

As you can see, the code between the `<<···>>` delimiters is not preprocessed: the double quotes and dollars are left as it. Moreover, every symbol is translated into the latex command `symbol` so it can be printed correctly. Spaces and new lines are also translated. To sum up, the text you write in verbatim mode will be printed verbatim[2].

Note that the `symbol` command of LaTeX often does not work well unless you use a `tt` font. The above example should be rewritten:

```
"Code: {texttt "<<printf "Toto has $%d to spare." 42>>"}"
```

This will produce:

Code: `printf "Toto has $%d to spare." 42`

Also note that the verbatim mode can only be used in text mode. If you write `<<` in math mode, this will produce $<<$, and in code mode this will produce a syntax error from the OCaml compiler or a quotation if you use the Camlp4 syntax extension.

The verbatim mode has other features such as delimiter selection, function selection and antiquotations. They will be described later in this document.

[1] It is *heavily* used in the source of this document, which is of course written using Melt.

[2] This is no coincidence.

2 The `Latex` Library

The `Latex` library is composed of a kernel, which allows to describe low-level LaTeX ASTs, and of basic function definitions built on top of this kernel.

2.1 Using the Library

2.1.1 Compiling the Library

The library is automatically compiled when you compile the Melt distribution. See its Makefile if you want to know more.

2.1.2 Installing the Library

The library will automatically be installed when you install the Melt distribution. It will normally be installed in the OCaml library directory, which you can find using the command `ocamlc -where`. The following files should be copied there:

```
latex.a
latex.cmi
latex.cma
latex.cmxa
```

To uninstall the library, simply delete these files.

2.1.3 Compiling a Program Which Uses the Library

At compile-time, the OCaml compiler needs to know where to find the interface `latex.cmi`. If this file has been installed in the default OCaml library directory, OCaml will find it automatically. Else, you need to include the directory using the `-I` option. For example, to compile `hello.ml` if the library has been installed in the `latex` subdirectory of the default directory:

```
ocamlc -c -I +latex hello.ml
```

At link-time, the OCaml compiler also needs to know that the archive `latex.cma`, along with its dependency `str.cma`, have to be linked with the program. Add all these archives in the command line, in the right order. Replace `.cma` with `.cmxa` if you use the native compiler. For example, to produce the `hello` bytecode executable:

```
ocamlc -I +latex str.cma latex.cma hello.cmo -o hello
```

2.2 The Kernel

2.2.1 Modes

2.2.2 Commands and Environments

2.2.3 Printing the AST

2.3 Pervasive Definitions

2.3.1 References and Labels

2.3.2 Verbatim

3 The Melt Preprocessor

- modes - pragmas - verbatim - command line options