Functional METAPOST for LaTeX[*]

Marco Kuhlmann[†]

2001/04/03

Abstract

Functional METAPOST (FMP) is a powerful frontend to the METAPOST language. This package adds basic FMP support to LaTeX, enabling users to keep FMP source code within their documents and, by a two-run mechanism, including automatically generated FMP figures.

1 Introduction

Functional METAPOST by Joachim Korittky ([Kor98]) adds a high-level interface to the METAPOST language ([Hob89], [Hob92]), enabling the user to program their graphics using the Haskell language. Impressive examples of Functional METAPOST can be found in Korittky's diploma thesis; some of them will be given below. The system and the documentation can be downloaded from

`http://www.informatik.uni-bonn.de/~ralf/software.html`

Using Functional METAPOST as my standard graphics developing tool, I felt a need to write a LaTeX package which smoothly integrates FMP into daily work, similar to the `emp` package by Thorsten Ohl ([Ohl97]); this is how `fmp` came to being. The earliest version supported only the possibility to automatically produce shell-scripts for graphics generation. Since then, I have added the possibility to encapsulate FMP code – though I still ask myself if this way of maintaining code is as natural for FMP as it is for pure METAPOST in `emp`.

In case you have any questions or comments on this package, feel free to send me an email. May `fmp` help FMP to spread around the world. `:-)`

1.1 Examples of Functional METAPOST

Before I start to present the `fmp` package, let me first give you two mouth-watering examples of FMP (the Haskell code for them can be found at the end of this document): figure 1 gives a binominal tree of rank 5, figure 2 shows a simple Venn diagram. Among other things, these examples exhibit two features in which FMP is superior to many other graphics drawing packages around:

- By embedding METAPOST into the Haskell programming language, FMP gives the user (and especially users who have previous experience in functional programming) a great tool to program graphics on a very high-level

[*]This file has version number v1.0.

[†]E-Mail: `marco.kuhlmann@gmx.net`

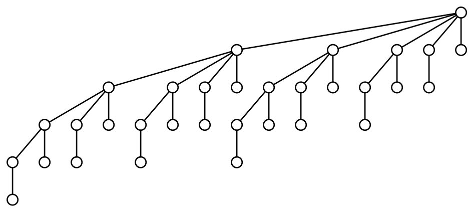

Figure 1: A binominal tree of rank 5

level of abstraction. This not only helps you to focus on the logical structure rather than on layout questions, but also is especially important if you want to scale and re-use old material.

- FMP can be easily extended. For example, for figure 1, the core language has been augmented by tree drawing features, using much better algorithms than those of any other tree drawing package around. This is especially good news for computer scientists, who need trees very often, but did not yet have a package to draw them on such a level of abstraction.

To be able to use FMP, you need a Haskell interpreter, such as `hugs`, and the METAPOST program, which should be part of any somewhat complete distribution of LaTeX. Having produced a Haskell source, you feed it into `hugs` and issue the `generate` command provided by FMP. This will translate your code into low-level METAPOST commands, and finally produce a ready-to use PostScript file.

1.2 How this package works

Calling `hugs` and typing in the generation commands is a tedious job if you keep more than but a few illustrations around. This package offers the `\fmpfigure` command, which generates a shell script (at present, this only works for Un∗x), which you then can execute to have all the graphics files generated at once. At the next run of LaTeX, these graphic files will appear at their proper positions. Besides, `fmp` enables you to store Haskell code within a LaTeX file, in case you want to have all the code for one document at one place.

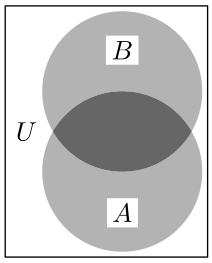

Figure 2: A Venn diagram (cf. figure 22 in [Hob92])

2 Usage

2.1 Including code

`fmp` If you want to store your FMP code in the same file than your LaTeX source, you can include it within the `fmp` environment. During compilation, the contents of the `fmp` environment are written to an external file (see below).

2.2 Including graphics

`\fmpfigure` You include a figure into your LaTeX document by using the `\fmpfigure` macro. It takes, as its only argument, the string identifying the figure in the Haskell source, and creates a shell script containing all the `hugs` calls needed for the actual generation. After the shell script has been written, you can execute it and run LaTeX again; if everything went right, the graphics file will appear at the place where you issued the `\fmpfigure` command.

2.3 Changing file names

`\fmpsourcefilename`
`\fmpscriptfilename`
`\fmpfigurebasename`

You can control the names of three different files:

- the Haskell source (`\fmpsourcefilename`, defaults to ⟨*jobname*⟩`.hs`),
- the shell script (`\fmpscriptfilename`, defaults to `fmpgenerate.sh`) and
- the graphics file base name (`\fmpfigurebasename`, defaults to `fmpfigure`), which is the base file name of the graphics files that will be generated by `hugs`. (An index number specifying their order in the document will be appended to this name.)

You can modify all three file names by calling the respective macros, each of which takes the new name as its argument.

2.4 Preamble and postamble

`\fmpsourcepreamble`
`\fmpaddtosourcepreamble`
`\fmpscriptpreamble`
`\fmpaddtoscriptpreamble`
`\fmpsourcepostamble`
`\fmpaddtosourcepostamble`
`\fmpscriptpostamble`
`\fmpaddtoscriptpostamble`

Before something is written to the source file or the shell script file, the package will output a *preamble* to that file. The source code preamble could contain everything from comments to Haskell module identifications and basic imports, while the shell script preamble should probably contain the line that calls `hugs` and tells it to input the rest of the file as comment. Have a sample run to see the default contents of the preamble. If you wish to change the text: you can set a new preamble by the `\fmpsourcepreamble` command, and you can append a new line to the current preamble calling `\fmpaddtosourcepreamble`. (Similar commands are available for the shell script preamble.) There is also a postamble, which is written as the very last thing to the source code or shell script.

2.5 Graphics file formats

The `graphicx` package is used to handle the inclusion of generated FMP figures. If `fmp` is called from within `pdflatex`, `graphicx` is loaded with the `pdftex` driver option. In this case, the fall-back behaviour when encountering an `\fmpfigure` command is to read the corresponding graphics file as `mps` (METAPOST output);

it will then internally be converted to PDF by `graphicx`. When called from within normal `latex`, graphics files are handled as `eps` (encapsulated PostScript). You probably need to load a specific PostScript driver for `graphicx` in this case – do so by supplying `fmp` with the same package option that you would use for `graphicx` (see the `graphicx` user manual for further information on that).

Source code for the examples

```
--
-- This is file ‘fmp-doc.hs’,
-- generated with the docstrip utility.
--
-- The original source files were:
--
-- fmp.dtx  (with options: ‘examples’)
--
-- Example source code for the FMP package
--
module FMPDoc where
import FMP
import FMPTree

example1          = binom 5
    where
    ce            = circle empty
    binom 0       = node ce []
    binom n       = node ce [edge (binom i)
                             | i <- [(n-1),(n-2)..0]]
                    #setAlign AlignRight

example2          = box (math "U" |||
                         ooalign [toPicture [cArea a 0.7,
                                             cArea b 0.7,
                                             cArea ab 0.4],
                                  bOverA])
    where
    cArea a c     = toArea a #setColor c
    bOverA        = column [math "B" #setBGColor white,
                            vspace 50,
                            math "A" #setBGColor white]
    a             = transformPath (scaled 30) fullcircle
    b             = transformPath (scaled 30 & shifted (0,-30))
                    fullcircle
    ab            = buildCycle a b
```

References

[Hob89] John D. Hobby: *A METAFONT-like System with PostScript Output.* TUGboat vol. 10, no. 2, pp. 505–512, 1989

[Hob92] JOHN D. HOBBY: *A User's manual for METAPOST.* Computing Science Technical Report no. 162, AT&T Bell Laboratories, Murray Hill, New Jersey, 1992

[Kor98] JOACHIM KORITTKY: *functional METAPOST. Eine Beschreibungssprache für Grafiken.* Diplomarbeit an der Rheinischen Friedrich-Wilhelms-Universität Bonn, 1998

[Ohl97] THORSTEN OHL: *EMP: Encapsulated METAPOST for LaTeX.* Technische Hochschule Darmstadt, 1998