Miscellaneous METAPOST Macros

Matej Košík

April 15, 2011

Abstract

Technical communication with people is critical, and harder than communicating with the machines. METAPOST is a high-level declarative and macro programming language. Due to its unusual domain (drawing pictures) bare code fragments often look like cryptic gibberish. In this document we try to prove that it is possible to write comprehensible METAPOST code[1].

Contents

Copyright and License

1 ⟨*misc.mp* 1⟩≡ 2a▷

```
% Additional macros for use with METAPOST
% Copyright (C) 2007 Matej Kosik <kosik@fiit.stuba.sk>
```

[1] Disclaimer: I do not explain METAPOST language here. You can refer for example to [1, 3].

```
%
% This program is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 2 of the License, or
% (at your option) any later version.
%
% This program is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License along
% with this program; if not, write to the Free Software Foundation, Inc.,
% 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
```

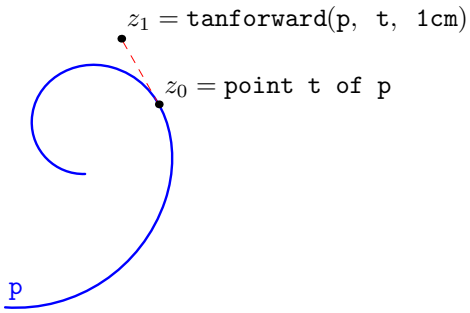
1 Various constants

2a ⟨*misc.mp* 1⟩+≡ ◁1 10a▷

```
color yellow, cyan;

yellow := red + green;
cyan := green + blue;
```

2 tanforward(expr p, t, d)

The purpose of this macro is illustrated on figure below:



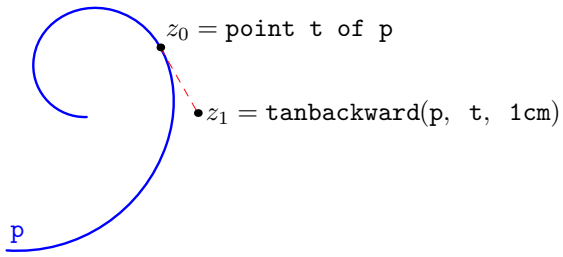
If we have a path `p` and we choose some time `t`, then z_0 are coordinates of a point along the curve `p` in time `t`. The `tanforward` macro computes coordinates of the point z_1. Point z_1 is on a tangent constructed in point z_0. It is in 1cm distance from point z_0.

2b ⟨*misc.mp* 2b⟩+≡ 3a▷

```
def tanforward(expr p, t, d) =
    (d,0) rotated (angle direction t of p) shifted (point t of p)
enddef;
```

3 tanbackward(expr p, t, d)

The `tanforward` macro returns coordinates along the tangent in positive direction, this `tanbackward` macro returns coordinates of a point in the opposite direction. See the figure below:

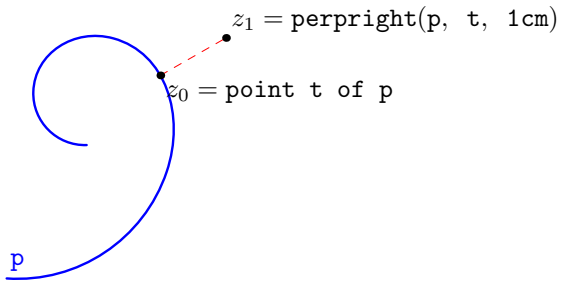


3a ⟨*misc.mp* 2b⟩+≡ ◁2b 3b▷

```
def tanbackward(expr p, t, d) =
    (d,0) rotated (180 + angle direction t of p) shifted (point t of p)
enddef;
```

4 perpright(expr p, t, d)

The purpose of this macro is illustrated on figure below:



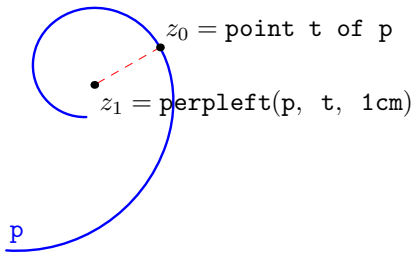
If we have a path `p` and we choose some time `t`, then z_0 are coordinates of a point along the curve `p` in time `t`. The `perpright` macro computes coordinates of the point z_1. Point z_1 is at the end of an abscissa beginning in point z_0 which has length `l` and it is perpendicular to the path `p`.

3b ⟨*misc.mp* 2b⟩+≡ ◁3a 4▷

```
def perpright(expr p, t, l) =
    (l,0) rotated ((angle direction t of p) - 90) shifted (point t of p)
enddef;
```

5 perpleft(expr p, t d)

It is very similar to `perpright` except coordinates in the opposite direction are returned. See the figure below:



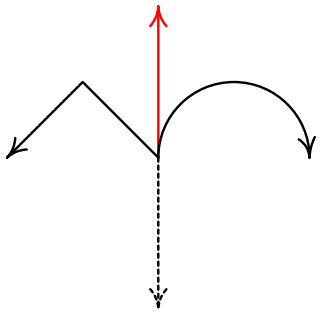
4 ⟨*misc.mp* 2b⟩+≡ ◁3b 5▷

```
def perpleft(expr p, t, l) =
    (l,0) rotated ((angle direction t of p) + 90) shifted (point t of p)
enddef;
```

6 drawkarrow expr path text t

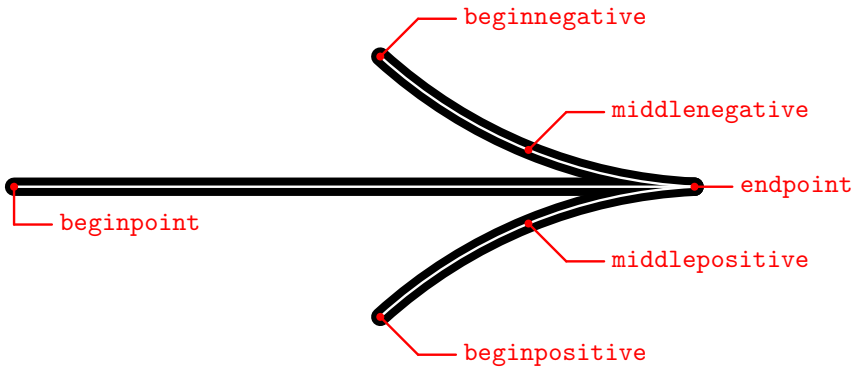
This macro draws an arrow similar to the arrows found in [2]. It can be used in the same way as the original `drawarrow` macro which is part of the *plain* format. Here are some example usages:

```
u := 1cm;
drawkarrow (0,0)--(-u,u)--(-2u,0);
drawkarrow (0,0)--(0,2u) withcolor red;
drawkarrow (0,0)..(u,u)..(2u,0);
drawkarrow (0,0)--(0,-2u) dashed withdots scaled .25;
```



That is, as you can see, the text after the `drawkarrow` has analogous meaning as the text after the original `drawarrow` macro. It only draws arrowhead in a different way.

Implementation of this macro is fairly delicate, so we will explain things in detail. Let us first enlarge the arrow and depict its distinct points:



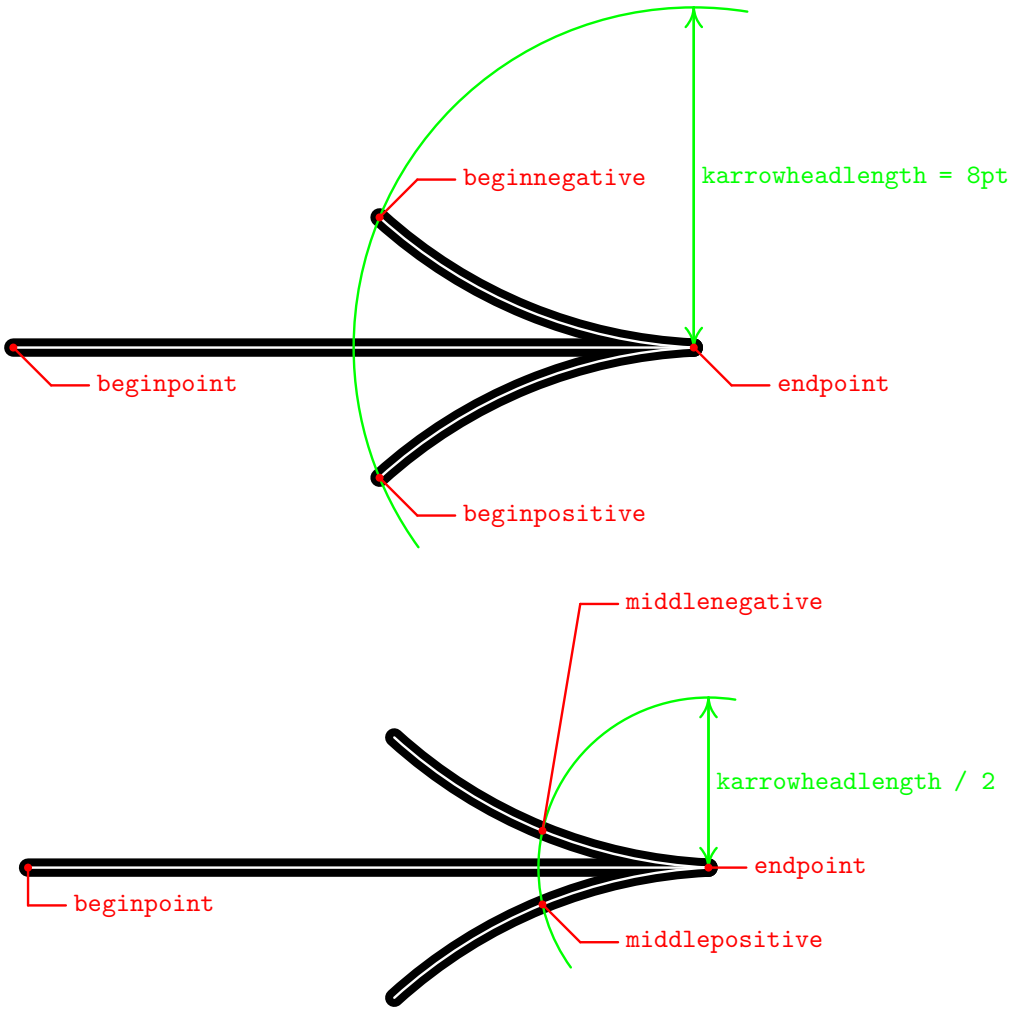
The arrow begins in `beginpoint` and ends in `endpoint`. These two points are are determined by the path which is given to this macro as a parameter. Coordinates of all the other points are computed.

The length of the arrowhead is influenced by the `karrowheadlength` variable. Its default value is 8 points.

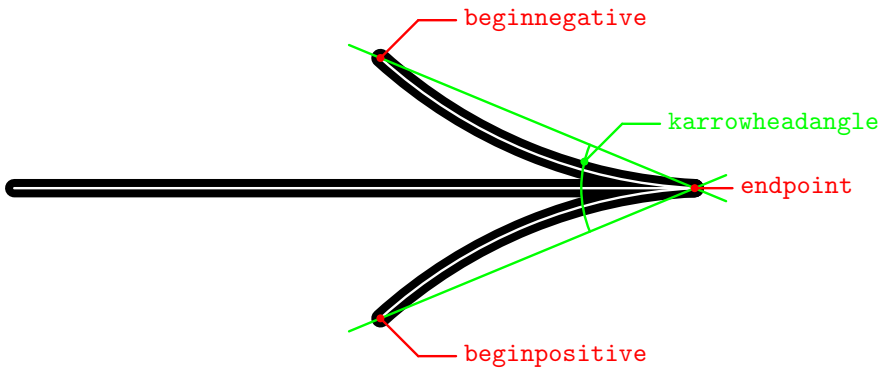
5 ⟨*misc.mp* 2b⟩+≡ ◁4 7a▷

```
karrowheadlength := 8pt;
```

The following figure illustrates the `karrowheadlength`.



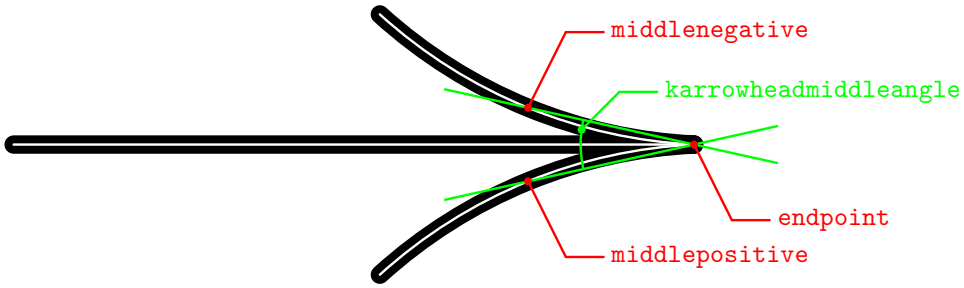
The `karrowheadangle` determines the angle shown in the figure below:



7a ⟨*misc.mp* 2b⟩+≡ ◁5 7b▷

```
karrowheadangle := 45;          % Default head angle of the end point.
```

The `karrowheadmiddleangle` determines the angle shown in the figure below:



7b ⟨*misc.mp* 2b⟩+≡ ◁7a 7c▷

```
karrowheadmiddleangle := 25;    % Default head angle of the middle point.
```

Auxiliary variables used for passing values between the main macro and its continuation.

7c ⟨*misc.mp* 2b⟩+≡ ◁7b 7d▷

```
path _mainpath, _positivearrowpath, _negativearrowpath;
```

7d ⟨*misc.mp* 2b⟩+≡ ◁7c 8▷

```
def drawkarrow expr p =
        begingroup
                save subp, endpoint, beginpoint, beginpositive, middlepositive,
             beginnegative, middlenegative;
        path subp;
        pair endpoint, beginpoint, beginpositive, middlepositive,
                 beginnegative, middlenegative;

        ⟨Compute _mainpath, _positivearrowpath and _negativearrowpath 9⟩
        endgroup;
    _drawkarrowcontinuation
enddef;
```

Continuation of the `drawkarrow` macro. It will use the computed `_mainpath`, `_positivearrowpath` and `_negativearrowpath`. It will also use (several times) the text which follows the path expression that follows the `drawkarrow` macro invocation. That is, consider the following statement:

```
drawkarrow (0,0)--(0,-2u) dashed withdots scaled .25;
```

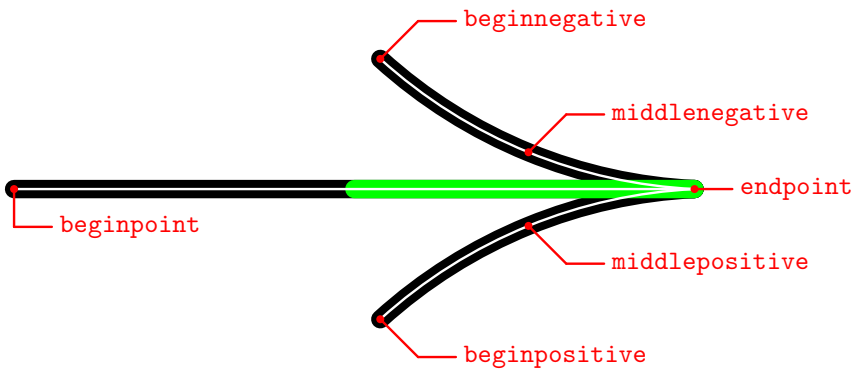
When `drawkarrow` macro is expanded, the `drawkarrow (0,0)--(0,-2u)` tokens are removed from the input stream. Immediately before invocation of the `_drawkarrowcontinuation`, the input stream looks as follows:

```
dashed withdots scaled .25;
```

And this rest (up to the nearest semicolon) is consumed by the `_drawkarrowcontinuation` macro and this text will be within this macro available under name `t`.

8 ⟨*misc.mp* 2b⟩+≡ ◁7d 10b▷

```
def _drawkarrowcontinuation text t =
    draw _mainpath t;
    draw _positivearrowpath t;
    draw _negativearrowpath t
enddef;
```



The code below computes the distinct points shown in the figure above. The value of `subp` path is denoted in green.

9 ⟨*Compute* `_mainpath`, `_positivearrowpath` *and* `_negativearrowpath` 9⟩≡ (7d)

```
endpoint := point length p of p;

subp := p cutbefore fullcircle
          scaled (2*karrowheadlength)
          shifted endpoint;

middlepositive :=
    point (.5*length subp)
    of (subp rotatedaround(endpoint, (.5*karrowheadmiddleangle)));

beginpositive :=
    point 0
    of (subp rotatedaround(endpoint, (.5*karrowheadangle)));

middlenegative :=
    point (.5*length subp)
    of (subp rotatedaround(endpoint, (-.5*karrowheadmiddleangle)));

beginnegative :=
    point 0
    of (subp rotatedaround(endpoint, (-.5*karrowheadangle)));

_mainpath := p;
_positivearrowpath := beginpositive..middlepositive..endpoint;
_negativearrowpath := beginnegative..middlenegative..endpoint;
```

7 drawtwowaykarrow expr path text t

This macro has a very similar effect as the `drawkarrow` except for that the arrowhead is drawn not only at the end of a given path but also at the beginning of that path. For example this code:

```
drawtwowaykarrow (0,0)..(7.5mm,7.5mm)..(15mm,0) withcolor red;
```

produces a following two-way arrow:

The implementation is lazy. We actually draw two k-arrows. On the original path and on the reversed path.

The value passed from the main macro to the continuation (the `_mainpath` identifier is already taken).

10a ⟨*misc.mp* 1⟩+≡ ◁2a

```
path __mainpath;
```

The main macro (which consumes the path expression) followed by its continuation which swallows the rest of the text up to the first semicolon.

10b ⟨*misc.mp* 2b⟩+≡ ◁8 11▷

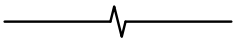
```
def drawtwowaykarrow expr p =
        __mainpath := p;
        drawtwowaykarrowcontinuation
enddef;

def drawtwowaykarrowcontinuation text t =
        drawkarrow __mainpath t;
        drawkarrow reverse __mainpath t
enddef;
```

8 drawinfinite expr path text t

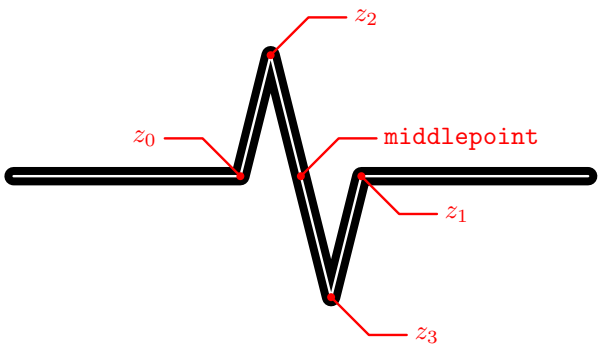
This macro can be used for drawing paths whose middle (mid-time) contains a mark which tells the reader that it is "infinite" or it was not drawn in its entirety. This macro can be used in a similar way as normal `draw` macro which is part of the *plain* format. With expression such as:

```
drawinfinite (0,0)--(3cm,0);
```

you can produce the following:



Let us enlarge the mark in the middle of this kind of path and denote distinct points:



11 ⟨*misc.mp* 2b⟩+≡ ◁10b 13▷

```
inflen := .2cm;                 % The length (along the line/curve) of the sign.
infwidth := .4cm;               % The width (perpendicularly to the line/curve).

% Auxiliary variables used for passing values between the main macro
% and its continuation.
path _mainpath, _firstpath, _secondpath;

def drawinfinite expr p =
    save middlepoint, middletime, tangent, pathlength,_middlepath,
         middlepathlength;
    pair middlepoint;
    path tangent, _middlenondrawncircle, _firsthalfpath, _secondhalfpath,
         _middlepath;
    save z; pair z[];

    pathlength := length p;
    middlepathlength := pathlength / 2;
    middlepoint := point middlepathlength  of p;
    _firsthalfpath := subpath (0, middlepathlength) of p;
    _secondhalfpath := subpath (middlepathlength, pathlength) of p;
    _middlenondrawncircle := fullcircle scaled inflen shifted middlepoint;
    z0 = _firsthalfpath intersectionpoint _middlenondrawncircle;
```

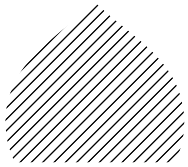
```
    z1 = _secondhalfpath intersectionpoint _middlenondrawncircle;
    _middlepath := z0--z1;
    middlepathlength := length _middlepath;
    z2 = perpleft(_middlepath, middlepathlength / 4, infwidth / 2);
    z3 = perpright(_middlepath, middlepathlength * 3 / 4, infwidth / 2);
    _middlepath := z0--z2--middlepoint--z3--z1;

    _drawinfinitecontinuation
enddef;

def _drawinfinitecontinuation text t =
    draw _firsthalfpath cutafter _middlenondrawncircle t;
    draw _middlepath t;
    draw _secondhalfpath cutbefore _middlenondrawncircle t
enddef;
```

9 drawhatched closedpath text t

This macro enables you to hatch interior of a given closed path. Example:

```
drawhatched (0,0)..(0,30)..{dir 45}(30,60){dir -45} ..(60,30)..(60,0)--cycle;
```

produces the following image



The text after the path expression influences the drawing of particular hatches. I.e.

```
drawhatched (0,0)..(0,30)..{dir 45}(30,60){dir -45} ..(60,30)..(60,0)--cycle
            withcolor red withpen pencircle scaled 2pt;
```

produces the following figure:

Auxiliary variables used for passing values between the main macro and its continuation. The underscore was prepended before their identifiers in the attempt to avoid name clashes[2].

13 ⟨*misc.mp* 2b⟩+≡ ◁11 14▷

```
path _pictureboundary;
picture _unclippedhatches;
path _boundingrectangle;
pair _lowerleft, _lowerright, _upperleft, _upperright;
```

[2]Which are, unfortunately, not excluded.

14 ⟨*misc.mp* 2b⟩+≡ ◁13

```
def drawhatched expr p =
    _unclippedhatches := nullpicture;
    _boundingrectangle := bbox p;

    % Reveal dimensions of the bounding rectangle of a given path.
    _lowerleft = llcorner _boundingrectangle;
    _lowerright = lrcorner _boundingrectangle;
    _upperleft = ulcorner _boundingrectangle;
    _upperright = urcorner _boundingrectangle;
    save width, height;
    width = xpart _lowerright - xpart _lowerleft;
    height = ypart _upperleft - ypart _lowerleft;

    % Find smallest square to which that rectangle fits.
    if width < height:
        _lowerright := _lowerleft + (height,0);
        _upperleft := _lowerleft + (0,height);
        _upperright := _lowerleft + (height,height);
        width := height;
    else:
        _lowerright := _lowerleft + (width,0);
        _upperleft := _lowerleft + (0,width);
        _upperright := _lowerleft + (width,width);
        height := width;
    fi;

    _pictureboundary := p;
    drawhatchedcontinuation
enddef;

def drawhatchedcontinuation text t =
    % Draw the hatches.
    for i=width step -.15cm until -width:
        addto _unclippedhatches contour _lowerleft+(0,i)--_upperright-(i,0)--cycle withpen penc
    endfor
    clip _unclippedhatches to _pictureboundary;
    draw _unclippedhatches
enddef;
```

References

[1] John D. Hobby. A User's Manual for METAPOST, April 1994.

[2] Donald E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley Publishing Co., Reading, Massachusetts, 2 edition, 1973.

[3] Urs Oswald. METAPOST: A Very Brief Tutorial, October 2002.