1. Preliminaries

{-# OPTIONS -W #-}
{-# LANGUAGE TypeSynonymInstances, FlexibleInstances, Rank2Types #-}

module Crossing where

import Control.Monad.State
import Data.Monoid (Monoid (.))
import Numeric (showFFloat)

We define a data type isomorphic to pairs, to express that one thing is above another spatially. For example, two threads might grow rightward in parallel, one above the other.

data above below = above below deriving Show

Given a list of things (such as threads ordered from top to bottom), we often want to pick out one or two of them by index and change those few while leaving the rest intact.

\[
at :: \text{Int} \rightarrow ([a] \rightarrow [a]) \rightarrow ([a] \rightarrow [a])
\]
\[
\text{at } i \ f \ \text{xs} = \text{ys} \oplus f \ \text{zs} \quad \text{where} \quad (\text{ys}, \text{zs}) = \text{splitAt } i \ \text{xs}
\]
\[
\text{at1} :: \text{Int} \rightarrow (a \rightarrow a) \rightarrow ([a] \rightarrow [a])
\]
\[
\text{at1 } i \ f = \text{at } i \ (\lambda (x : zs). f \ x : zs)
\]
\[
\text{at2} :: \text{Int} \rightarrow (\langle a, a \rangle \rightarrow \langle a, a \rangle) \rightarrow ([a] \rightarrow [a])
\]
\[
\text{at2 } i \ f = \text{at } i \ (\lambda (x : y : zs). \text{let } x' = f \ x \ y \ \text{in } x' : y : zs)
\]

2. Weaving

Weaving is basically the process of taking a list of crossings and following those commands to permute a list of threads. A crossing is basically an command to swap two adjacent threads, so the two threads to swap are identified by a single integer index into the list of threads. A crossing is either positive (right-handed, \(\times\)) or negative (left-handed, \(\times\)).

data \(PN = \times \mid \times\) deriving Show

Each thread records the history of how it has been swapped with other threads. The history is a list of \(\text{Signals}\), one per swap experienced by the thread (from most recent to most ancient—that is, in reverse order).

data \(\text{Signal } c a = \text{Signal } OU \mid c a\) deriving Show

data \(\text{Thread } c a t = \text{Thread } t [\text{Signal } c a]\) deriving Show

instance Functor (\(\text{Thread } c a t\)) where fmap f (\(\text{Thread } t \text{ss}\)) = \(\text{Thread } (f \ t) \text{ss}\)

The most important thing that a \(\text{Signal}\) records is whether the thread crossed \(\text{under}\) another thread \(\text{\textbullet}\) or not \(\text{\rightarrow}\). This information has type \(OU\). At each crossing, one thread goes over the other, depending on the sign of the crossing.

data \(OU = \text{\textbullet} \mid \text{\rightarrow}\) deriving (\(\text{Eq}, \text{Show}\))

\(ou\text{\textbullet}, ou\text{\rightarrow} :: PN \rightarrow OU\)
\(ou\text{\textbullet} \times = \text{\textbullet}\)
\(ou\text{\rightarrow} \times = \text{\rightarrow}\)
A crossing can also contain information such as its location and the tangent directions of the two threads. Between the histories of the two threads, the location of the crossing is shared whereas the tangent directions are split. In other words, each Signal records one crossing location (the same between the two threads, of type c) and one tangent direction (different between the two threads, of type a).

Actually, weaving operates not on a list of Threads but on a list of Maybe Threads. The reason is that sometimes we want to imagine that a thread is there (and swap it with other threads) but not draw it yet. We represent such an imaginary thread by Nothing. When a real thread crosses an imaginary thread, the real thread always goes over (→), so no gap is drawn. (The first state component below, of type c, is explained shortly with advance.)

\[
\text{type } M \: c \: a \: t = \text{State} (c, [\text{Maybe (Thread } c \: a \: t)])
\]

\[
crossing :: \text{Monoid } c \Rightarrow \text{Int} \rightarrow \text{PN} \rightarrow c \rightarrow a \rightarrow M \: c \: a \: t ()
\]

\[
crossing i \: pn \: c \: a = \text{modify} (\lambda (c_0, \text{threads}). (c_0, \text{at2 } i \: f \: c_0 \: \text{threads}))
\]

Furthermore, weaving actually processes not a just list of crossings but more generally a list of commands. Crossings are by far the most common kind of commands, but there are many other kinds.

An advance command shifts all future crossing locations by the specified amount, as if that much space has been consumed by the weaving. The amount to shift is the first state component in the M monad.

\[
\text{advance :: } \text{Monoid } c \Rightarrow c \rightarrow M \: c \: a \: t ()
\]

\[
\text{advance } c = \text{modify} (\lambda (c_0, \text{threads}). (c_0 \circ c, \text{threads}))
\]

A through command forces a thread to go through a location without crossing any other thread.

\[
\text{through :: } \text{Monoid } c \Rightarrow \text{Int} \rightarrow c \rightarrow a \rightarrow M \: c \: a \: t ()
\]

\[
\text{through } i \: c \: a = \text{modify} (\lambda (c_0, \text{threads}). (c_0 \circ \text{at1 } i \: (fmap \: (f \: c_0)) \: \text{threads}))
\]

A begin command turns an imaginary thread into a real one—in other words, puts the pen down. The second argument to begin (of type t) specifies information about the new real thread such as its identity and stroke color and whether its two ends should be connected to form a loop.

\[
\text{begin :: } \text{Eq } t \Rightarrow \text{Int} \rightarrow t \rightarrow M \: c \: a \: t ()
\]

\[
\text{begin } i \: t = \text{modify} (\lambda (c_0, \text{threads}). (c_0 \circ \text{at1 } i \: f \: \text{threads}))
\]
\[ f \text{(Just \_)} = \text{error ("Thread already begun at "+ show i)} \]
\[ f \text{Nothing} = \text{Just (Thread t [])} \]

Dually, an \text{end} command raises the pen, by moving a real thread to the end of the list and putting an imaginary thread where the real thread was.

\[ \text{end :: Eq t \Rightarrow Int \rightarrow t \rightarrow M c a t (\_)} \]
\[ \text{end i t = modify} \lambda(c_0, \text{threads}).(c_0, f (\text{splitAt i threads})) \]
\[ \text{where} \]
\[ f \text{(above, thread@Just (Thread t' \_)): below)} \]
\[ | t \equiv t' = \text{above} + \text{Nothing} : \text{below} + [\text{thread}] \]
\[ | \text{otherwise} \quad \text{= error ("Different thread begun at "+ show i)} \]
\[ f \text{(_, Nothing): _} = \text{error ("Thread never begun at "+ show i)} \]
\[ f \text{(_, [])} = \text{error ("Thread index "+ show i +" out of range")} \]

To carry out a command, we typically start with no shift and a list full of imaginary threads, and do not care about the final shift.

\[ \text{weave' :: Monoid c \Rightarrow M c a t (\_)} \]
\[ \text{weave' \Rightarrow execState} \]
\[ \text{weave :: Monoid c \Rightarrow M c a t (\_) \rightarrow Int \rightarrow [Maybe (Thread c a t)]} \]
\[ \text{weave cmd nThreads = snd (weave' cmd (mempty, replicate nThreads Nothing))} \]

3. Space

Now we are ready to weave some threads in 2D. First we need some basic functions on 2D vectors.

\text{type Coord = Double} \]
\text{data Coords = \langle Coord, Coord\rangle} \]
\text{infixl 6 \oplus, \ominus} \]
\text{infixl 7 \otimes, \oslash} \]
\[ \langle x_1, y_1 \rangle \oplus \langle x_2, y_2 \rangle = \langle x_1 + x_2, y_1 + y_2 \rangle \]
\[ \langle x_1, y_1 \rangle \ominus \langle x_2, y_2 \rangle = \langle x_1 - x_2, y_1 - y_2 \rangle \]
\[ \langle x, y \rangle \otimes t = \langle x \times t, y \times t \rangle \]
\[ \langle x, y \rangle \oslash t = \langle x/t, y/t \rangle \]
\[ \ominus, \text{normalize} \quad : \text{Coords \rightarrow Coords} \]
\[ \ominus \langle x, y \rangle = \langle \text{negate} x, \text{negate} y \rangle \]
\[ \text{normalize} z = z \oslash |z| \]
\[ |\cdot| \quad : \text{Coords \rightarrow Coord} \]
\[ |\langle x, y \rangle| = \sqrt{x \times x + y \times y} \]

\text{instance Show Coords where} \]
\[ \text{showsPrec p \langle x, y \rangle = showParen (p > 10) (s x \circ showChar ' ', \circ s y)} \]
\[ \text{where s} = \text{showFFloat (Just 5)} \]
instance Monoid Coords where
  mempty = ⟨0, 0⟩
  · ⋄ · = (⊕)

The following function \( \text{arg} \) takes two vectors as arguments and computes the cos and sin of the angle counterclockwise from the second vector to the first.

\[
\text{arg} :: \text{Coords} \rightarrow \text{Coords} \rightarrow \text{Coords}
\]

\[
\text{arg} \langle x_1, y_1 \rangle \langle x_2, y_2 \rangle = \text{normalize} \langle x_1 \times x_2 + y_1 \times y_2, y_1 \times x_2 - x_1 \times y_2 \rangle
\]

As promised, we record at each crossing the location and the tangent directions of the two threads. We record the pen color of each thread as a string. We also record whether the two ends of each thread should be connected to form a loop.

\[
\begin{align*}
\text{type } M \_ t &= M \text{ Coords Coords } t \\
\text{type } \text{Signal} \_ &= \text{Signal} \text{ Coords Coords} \\
\text{type } \text{Thread} \_ &= \text{Thread} \text{ Coords Coords} \\
\text{data } \text{OC} &= \text{Open} \mid \text{Closed} \quad \text{deriving} \ (\text{Eq}, \text{Show}, \text{Read}) \\
\text{data } \text{Paint} &= \text{Paint} \text{ String } \text{OC} \quad \text{deriving} \ (\text{Eq}, \text{Show}, \text{Read}) \\
\text{paint, red, orange} &= : \text{OC} \rightarrow \text{Paint} \\
paint &= \text{Paint } "" \\
\text{red} &= \text{Paint } "\text{red}" \\
\text{orange} &= \text{Paint } "\text{orange}" \\
\end{align*}
\]

We draw a thread by alternating between cubic Bézier segments (between crossings) and straight line segments (at crossings; only if \( \rightarrow \), not if \( \rightarrow \)). The Bézier control points are chosen as by Hobby (1986). The length of the straight line segment is twice the length of the tangent direction vector specified. At each of the two ends of a thread is an additional line segment, whose length is \( \text{stub} \).

\[
\begin{align*}
\text{stub} &= \text{Coord} \\
\text{stub} &= 5 \\
\text{hobby} &= \text{Coords} \rightarrow \text{Coords} \rightarrow \text{Coords} \rightarrow \text{Coords} \rightarrow (\text{Coords}, \text{Coords}) \\
\text{hobby } z_0 z_1 w_0 w_1 &= (z_0 \oplus w_0 \otimes (\rho/(3 \times \tau_0) \times n/|w_0|)), \\
&\quad z_1 \oplus w_1 \otimes (\sigma/(-3 \times \tau_1) \times n/|w_1|)) \quad \text{where} \\
\end{align*}
\]

\[
\begin{align*}
d &= z_1 \ominus z_0 \\
n &= |d| \\
\langle \cos \theta, \sin \theta \rangle &= \text{arg } w_0 d \\
\langle \cos \phi, \sin \phi \rangle &= \text{arg } d w_1 \\
a &= \sqrt{2} \\
b &= 1/16 \\
c &= (3 - \sqrt{5})/2 \\
\alpha &= a \times (\sin \theta - b \times \sin \phi) \times (\sin \phi - b \times \sin \theta) \times (\cos \theta - \cos \phi) \\
\rho &= (2 + \alpha)/(1 + (1 - c) \times \cos \theta + c \times \cos \phi) \\
\sigma &= (2 - \alpha)/(1 + (1 - c) \times \cos \phi + c \times \cos \theta) \\
\tau_0 &= 1 \\
\tau_1 &= 1
\end{align*}
\]
It is handy to transform thread coordinates by a function from $\mathbb{R}^2$ to $\mathbb{R}^2$. To transform tangent directions along with crossing locations, we need the function to return its own partial derivatives.

\[
\text{transform} :: (\text{Coords} \rightarrow (\text{Coords}, \text{Coords}, \text{Coords})) \rightarrow [\text{Signal}] \rightarrow [\text{Signal}]
\]

\[
\text{transform } f \; \text{ss} = [\text{Signal } \text{ou } \text{cc} (a' \otimes (|a|/|a'|))
| \text{Signal } \text{ou } (ax, ay) \leftarrow \text{ss},
\text{let } (cc, c'x, c'y) = f \; c
a' = c'x \otimes ax \oplus c'y \otimes ay]
\]

We can transform a picture on a cylinder into a picture on an annulus, to see the final result of a construction. To this end, the \textit{circular} command executes a subcommand and transforms the result so that the threads go around a circle and their ends connect to each other according to their initial and final positions in the thread list. In other words, the threads are concatenated according to how their end points are identified along the seam of the cylinder. The circumference of the circle and the phase along the circle are determined by the total \textit{advance} amount in the subcommand: the straight line segment between the origin and the total \textit{advance} amount will be curled up into a circle while preserving the total length and the initial and final tangent direction.

\[
\text{circular} :: (\forall t'. M_\rightarrow t') \rightarrow M_\rightarrow t
\]

\[
\text{circular cmd} = \text{modify } (\lambda (c_0, \text{threads}). (c_0, \text{zipWith } (g \; c_0) \; \text{threads } [0..]))
\text{where } (c_1, \text{threads1}) = \text{weave'} \; \text{cmd } (\text{mempty}, [\text{Just } (\text{Thread } i [\]) | i \leftarrow [0..]])
\]

\[
g \; \text{Nothing } = \text{Nothing}
\]

\[
g \; \text{c_0 } (\text{Just } (\text{Thread } t \; \text{ss})) \; i = \text{Just } (\text{Thread } t \; (\text{transform } (f \; c_0) \; (\text{loop } i) \oplus \text{ss}))
\text{loop } i = \text{ss } \oplus \text{go } j
\text{where } \text{Just } (\text{Thread } j \; \text{ss}) = \text{threads1 } ! i
\text{go } j | i \equiv j = []
\text{otherwise } = \text{ss } \oplus \text{go } k
\text{where } \text{Just } (\text{Thread } k \; \text{ss}) = \text{threads1 } ! j
\]

\[
p = |c_1|
\]

\[
\langle rc, rs \rangle = c_1 \otimes p
\]

\[
r_0 = p/(2 \times \pi)
\]

\[
f \; c_0 \; \langle x, y \rangle = \text{let } \langle x', y' \rangle = \langle rc, -rs \rangle \otimes x \oplus \langle rs, rc \rangle \otimes y
\]

\[
t = x'/p
\]

\[
r = y' + r_0
\]

\[
s = \sin (2 \times \pi \times t)
\]

\[
c = \cos (2 \times \pi \times t)
\]

\[
x'' = r \times s
\]

\[
y'' = r \times c - r_0
\]

\[
d \; \langle dx, dy \rangle = \text{let } \langle dx', dy' \rangle = \langle rc, -rs \rangle \otimes dx \oplus \langle rs, rc \rangle \otimes dy
\]

\[
dt = dx'/|c_1|
\]

\[
dr = dy'
\]

\[
ds = 2 \times \pi \times dt \times c
\]

\[
dc = -2 \times \pi \times dt \times s
\]

\[
dx'' = dr \times s + r \times ds
\]
\[ dy'' = dr \times c + r \times dc \]
\[ \text{in} (rc, rs) \otimes dx'' \oplus (-rs, rc) \otimes dy'' \]
\[ \text{in} (c_0 \oplus (rc, rs) \otimes x'' \oplus (-rs, rc) \otimes y'', d \langle 1, 0 \rangle, d \langle 0, 1 \rangle) \]

We use rectangles to track the bounding box of a picture.

\text{data Rect} = Rect Coords Coords  
\text{deriving Show}

\text{data Bounds} = Empty \mid \text{Nonempty Rect}  
\text{deriving Show}

\text{instance Monoid Bounds where}
\text{mempty} = Empty
\text{Empty} \circ b = b
b \circ \text{Empty} = b
\text{Nonempty} (\text{Rect} \langle x_1, y_1 \rangle \langle x_2, y_2 \rangle) \circ \text{Nonempty} (\text{Rect} \langle x'_1, y'_1 \rangle \langle x'_2, y'_2 \rangle) = \text{Nonempty} (\text{Rect} \langle \min x_1 x'_1, \min y_1 y'_1 \rangle \langle \max x_2 x'_2, \max y_2 y'_2 \rangle)

\text{class HasBounds a where}
\text{bounds} :: a \rightarrow \text{Bounds}

\text{instance HasBounds Signal where}
\text{bounds} (\text{Signal} c a) = \text{Nonempty} (\text{Rect} (c \ominus d) (c \oplus d))
\text{where} \langle dx, dy \rangle = a \otimes (1 + \text{stub} / |a|)
d = \langle \text{abs} dx, \text{abs} dy \rangle

\text{instance HasBounds (Thread \_ t) where}
\text{bounds} (\text{Thread} \_ \text{signals}) = \text{bounds signals}

\text{instance HasBounds a \Rightarrow HasBounds [a] where}
\text{bounds} = \text{maybe Empty bounds}

\text{instance HasBounds a \Rightarrow HasBounds (Maybe a) where}
\text{bounds} = \text{maybe Empty bounds}

4. SVG output

The function \text{path} draws a thread history as an SVG path. If the first argument is \text{Closed}, then \text{path} draws a closed thread (which requires a closed SVG path if and only if there is no gap (\(-\)) in the thread).

\text{path} :: \text{OC} \rightarrow [\text{Signal} \_] \rightarrow \text{String}
\text{path Open} [] = ""
\text{path Open} (\text{Signal} ou c a : ss) =
'\text{M}': \text{show} (c \oplus a \otimes (1 + \text{stub} / |a|)) +
\text{case ou of} \rightarrow \text{path}' \rightarrow c a ss \text{True}
-\rightarrow 'L': \text{show} (c \oplus a) \oplus \text{path}' \rightarrow c a ss \text{True}
\text{path Closed} ss =
\text{case span} (\lambda(Signal ou \_). ou \equiv \rightarrow) ss \text{ of}
(\text{[]}, []) \rightarrow ""
(s@\text{Signal} \rightarrow c a : \text{rest}, []) \rightarrow \text{path}' \rightarrow c a (\text{rest} \oplus [s]) \text{False} \oplus ['Z']
(\text{initial}, s@\text{Signal} \rightarrow c a : \text{rest}) \rightarrow \text{path}' \rightarrow c a (\text{rest} \oplus \text{initial} \oplus [s]) \text{False}
-\rightarrow \text{error} "\text{Internal error: unexpected pattern-match failure}"

\text{path'} :: \text{OU} \rightarrow \text{Coords} \rightarrow \text{Coords} \rightarrow [\text{Signal} \_] \rightarrow \text{Bool} \rightarrow \text{String}
The following function draws a bunch of threads as an SVG document.

\[
\text{svg} :: [\text{Maybe (Thread \_ Paint)}] \rightarrow \text{String}
\]

\[
\text{svg threads = unlines}$\begin{verbatim}$
\text{("<svg xmlns="http://www.w3.org/2000/svg" width="" + show width + \\
" height="" + show height + "">")}
: ("<g transform="scale(5 -5) translate(" + show tx + " " + show ty + ")"></")
: [ "<path fill="none" stroke="" + stroke + \\
" stroke-width=".8" d="" + path oc ss + ""/>"
| Just (Thread (Paint paint oc) ss) \leftarrow threads,
let stroke = if null paint then "currentColor" else paint]
+ ["</g></svg>" ]
\end{verbatim}$
\[
\text{where scale } = 5
\]
\[
\text{margin } = 2
\]
\[
((tx, ty), (width, height)) =
\text{case bounds threads of}
\]
\[
\text{Empty } \rightarrow ((0, 0), (0, 0))
\]
\[
\text{Nonempty } (\text{Rect } \langle x_1, y_1 \rangle \langle x_2, y_2 \rangle) \rightarrow
((\text{margin } - x_1, -\text{margin } - y_2),
(x_2 - x_1 + 2 \times \text{margin}, y_2 - y_1 + 2 \times \text{margin}) \otimes \text{scale})
\]

We can also put the same picture in TikZ.

\[
\text{tikz} :: [\text{Maybe (Thread \_ Paint)}] \rightarrow \text{String}
\]

\[
\text{tikz threads = unlines}$\begin{verbatim}$
\text{\begin{tikzpicture}[line width=.8pt]}
\text{\draw } \begin{verbatim}$
\text{\" + options + "svg \" + path oc ss + \\
\~/d="" + path oc ss + ""/>"
| Just (Thread (Paint paint oc) ss) \leftarrow threads,
\text{let options = if null paint then "" else "[draw=" + paint + "] " ]}
\end{verbatim}$
\text{\end{tikzpicture}%}$
\end{verbatim}$
\[
\text{\begin{verbatim}$
\text{where scale } = 5
\]
\[
\text{margin } = 2
\]
\[
((tx, ty), (width, height)) =
\text{case bounds threads of}
\]
\[
\text{Empty } \rightarrow ((0, 0), (0, 0))
\]
\[
\text{Nonempty } (\text{Rect } \langle x_1, y_1 \rangle \langle x_2, y_2 \rangle) \rightarrow
((\text{margin } - x_1, -\text{margin } - y_2),
(x_2 - x_1 + 2 \times \text{margin}, y_2 - y_1 + 2 \times \text{margin}) \otimes \text{scale})
\]

References

Hobby, John D. 1986. Smooth, easy to computer interpolating splines. Discrete and Com-