Pointfree

From HaskellWiki

Contents

Pointfree Style

It is very common for functional programmers to write functions as a composition of other functions, never mentioning the actual arguments they will be applied to. For example, compare:

```
sum = foldr (+) 0
```

with:

```
sum' xs = foldr (+) 0 xs
```

These functions perform the same operation, however, the former is more compact, and is considered cleaner. This is closely related to function pipelines (and to unix shell scripting (http://www.vex.net/~trebla/weblog/pointfree.html)): it is clearer to write

```
let fn = f . g . h
```

than to write

```
let fn x = f (g (h x))
```

.

This style is particularly useful when deriving efficient programs by calculation and, in general, constitutes good discipline. It helps the writer (and reader) think about composing functions (high level), rather than shuffling data (low level).

It is a common experience when rewriting expressions in pointfree style to derive more compact, clearer versions of the code -- explicit points often obscure the underlying algorithm.

Point-free map fusion:

```
foldr f e . map g == foldr (f . g) e
```

versus pointful map fusion:

```
foldr f e . map g == foldr f' e
     where f' a b = f (g a) b
```

Some more examples:

```
-- point-wise, and point-free member
mem, mem' :: Eq a => a -> [a] -> Bool

mem x lst = any (== x) lst
mem'      = any . (==)
```

1 But pointfree has more points!

A common misconception is that the 'points' of pointfree style are the `(.)` operator (function composition, as an ASCII symbol), which uses the same identifier as the decimal point. This is wrong. The term originated in topology, a branch of mathematics which works with spaces composed of points, and functions between those spaces. So a 'points-free' definition of a function is one which does not explicitly mention the points (values) of the space on which the function acts. In Haskell, our 'space' is some type, and 'points' are values. In the declaration `f x = x + 1` we define the function `f` in terms of its action on an arbitrary point `x`. Contrast this with the points-free version: `f = (+ 1)`

where there is no mention of the value on which the function is acting.

2 Background

To find out more about this style, search for Squiggol and the Bird-Meertens Formalism, a style of functional programming by calculation that was developed by Richard Bird (http://web.comlab.ox.ac.uk/oucl/work/richard.bird/publications.html) , Lambert Meertens (http://www.kestrel.edu/home/people/meertens/) , and others at Oxford University. Jeremy Gibbons

(http://web.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/) has also written a number of papers about the topic, which are cited below.

3 Tool support

Thomas Yaeger has written (http://www.cse.unsw.edu.au/~dons/code/lambdabot/Plugins/Pl/) a Lambdabot (http://haskell.org/haskellwiki/Lambdabot) plugin to automatically convert a large subset of Haskell expressions to pointfree form. This tool has made it easier to use the more abstract pointfree encodings (as it saves some mental gymnastics on the part of the programmer). You can experiment with this in the Haskell IRC channel. A stand-alone command-line version is available at HackageDB (http://hackage.haskell.org/package/pointfree) (package pointfree).

The @pl (point-less) plugin is rather infamous for using the
`(->) a`
monad to obtain concise code. It also makes use of Arrows. It also sometimes produces (amusing) code blow ups with the
`(.)`
operator.

Recently, @unpl has been written, which (attempts) to unscramble @pl-ified code. It also has a stand-alone command-line version (http://hackage.haskell.org/package/pointful) (package pointful).

A transcript:

```
> pl \x y -> x y
id

> unpl id
(\ a -> a)

> pl \x y -> x + 1
const . (1 +)

> unpl const . (1 +)
(\ e _ -> 1 + e)

> pl \v1 v2 -> sum (zipWith (*) v1 v2)
(sum .) . zipWith (*)

> unpl (sum .) . zipWith (*)
(\ d g -> sum (zipWith (*) d g))

> pl \x y z -> f (g x y z)
((f .) .) . g

> unpl ((f .) .) . g
(\ e j m -> f (g e j m))

> pl \x y z -> f (g x y) z
(f .) . g
```

```
> unpl (f .) . g
(\ d i -> f (g d i))

> pl \x y z -> f z (g x y)
(flip f .) . g

> unpl (flip f .) . g
(\ i l c -> f c (g i l))

> pl \(a,b) -> (b,a)
uncurry (flip (,))

> pl f a b = b a
f = flip id

> pl \ x -> x * x
join (*)

> pl \a b -> a:b:[]
(. return) . (:)

> pl \x -> x+x+x
(+) =<< join (+)

> pl \a b -> Nothing
const (const Nothing)

> pl \(a,b) -> (f a, g b)
f *** g

> pl \f g h x -> f x `h` g x
flip . (ap .) . flip (.)

> pl \x y -> x . f . y
(. (f .)) . (.)

> pl \f xs -> xs >>= return . f
fmap

> pl \h f g x -> f x `h` g x
liftM2

> pl \f a b c d -> f b c d a
flip . ((flip . (flip .)) .)

> pl \a (b,c) -> a c b
(`ap` snd) . (. fst) . flip

> pl \x y -> compare (f x) (f y)
((. f) . compare .)
```

For many many more examples, google for the results of '@pl' in the #haskell logs. (Or join #haskell on FreeNode and try it yourself!) It can, of course, get out of hand:

```
> pl \(a,b) -> a:b:[]
uncurry ((. return) . (:))

> pl \a b c -> a*b+2+c
```

```
((+) .) . flip flip 2 . ((+) .) . (*)

> pl \f (a,b) -> (f a, f b)
(`ap` snd) . (. fst) . (flip =<< (((.) . (,)) .))

> pl \f g (a,b) -> (f a, g b)
flip flip snd . (ap .) . flip flip fst . ((.) .) . flip . (((.) . (,)) .)

> unpl flip flip snd . (ap .) . flip flip fst . ((.) .) . flip . (((.) . (,)) .)
(\ aa f ->
  (\ p w -> ((,)) (aa (fst p)) (f w)) >>=
     \ ao -> snd >>= \ an -> return (ao an))
```

4 Combinator discoveries

Some fun combinators have been found via @pl. Here we list some of the best:

4.1 The owl

```
((.)$(.))
```

The owl has type
`(a -> b -> c) -> a -> (a1 -> b) -> a1 -> c`
, and in pointful style can be written as
`f a b c d = a b (c d)`
.

Example

```
>  ((.)$(.)) (==) 1 (1+) 0
True
```

4.2 Dot

```
dot = ((.).(.))

dot :: (b -> c) -> (a -> a1 -> b) -> a -> a1 -> c
```

Example:

```
sequence `dot` replicate ==
(sequence .) . replicate ==
replicateM

(=<<) == join `dot` fmap
```

4.3 Swing

-- Note: @pl had nothing to do with the invention of this combinator. I constructed it by hand after noticing a common pattern. -- Cale

```
swing :: (((a -> b) -> b) -> c -> d) -> c -> a -> d
swing = flip . (. flip id)
```

```
swing f = flip (f . runCont . return)
swing f c a = f ($ a) c
```

Some examples of use:

```
swing map :: forall a b. [a -> b] -> a -> [b]
swing any :: forall a. [a -> Bool] -> a -> Bool
swing foldr :: forall a b. b -> a -> [a -> b -> b] -> b
swing zipWith :: forall a b c. [a -> b -> c] -> a -> [b] -> [c]
swing find :: forall a. [a -> Bool] -> a -> Maybe (a -> Bool)
   -- applies each of the predicates to the given value, returning the first predicate which
swing partition :: forall a. [a -> Bool] -> a -> ([a -> Bool], [a -> Bool])
```

4.4 Squish

```
f >>= a . b . c =<< g
```

Example:

```
(readFile y >>=) . ((a . b) .) . c =<< readFile x
```

An actually useful example, numbering lines of a file.

5 Problems with pointfree

Point-free style can (clearly) lead to Obfuscation when used unwisely. As higher-order functions are chained together, it can become harder to mentally infer the types of expressions. The mental cues to an expression's type (explicit function arguments, and the number of arguments) go missing.

Point-free style often times leads to code which is difficult to modify. A function written in a pointfree style may have to be radically changed to make minor changes in functionality. This is because the function becomes more complicated than a composition of lambdas and other functions, and compositions must be changed to application for a pointful function.

Perhaps these are why pointfree style is sometimes (often?) referred to as *pointless style*.

6 References

One early reference is

- Backus, J. 1978. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," Communications of the Association for Computing Machinery 21:613-641.

which appears to be available (as a scan) at http://www.stanford.edu/class/cs242

/readings/backus.pdf

A paper specifically about point-free style:

- http://web.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/index.html#radix

This style underlies a lot of expert Haskeller's intuitions. A rather infamous paper (for all the cute symbols) is Erik Meijer et. al's:

- Functional Programming with Bananas, Lenses, and Barbed Wire, http://wwwhome.cs.utwente.nl/~fokkinga/mmf91m.ps.

Squiggol (http://en.wikipedia.org/wiki/Squiggol) , and the Bird-Meertens Formalism:

- http://web.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/index.html#squiggolintro.
- A Calculus of Functions for Program Derivation, R.S. Bird, in Res Topics in Fnl Prog, D. Turner ed, A-W 1990.
- The Squiggolist, ed Johan Jeuring, published irregularly by CWI Amsterdam.

Pointless Haskell (http://wiki.di.uminho.pt/twiki/bin/view/Personal/Alcino/PointlessHaskell) is a library for point-free programming with recursion patterns defined as hylomorphisms. It also allows the visualization of the intermediate data structure of the hylomorphisms with GHood. This feature together with the DrHylo tool allows us to easily visualize recursion trees of Haskell functions. Haskell Manipulation (http://wiki.di.uminho.pt/wiki/pub/Ze/Bic/report.pdf) by Jose Miguel Paiva Proenca discusses this tool based approach to re-factoring.

This project is written by Manuel Alcino Cunha (http://www.di.uminho.pt/~mac/) , see his homepage for more related materials on the topic. An extended verson of his paper *Point-free Programming with Hylomorphisms* can be found here (http://web.comlab.ox.ac.uk/oucl/research/pdt/ap/dgp/workshop2004/cunha.pdf) .

7 Other areas

Combinatory logic and also Recursive function theory can be said in some sense pointfree.

Are there pointfree approaches to relational algebra? See First Steps in Pointfree Functional Dependency Theory (http://www.di.uminho.pt/~jno/ps/_.pdf) written by José Nuno Oliveira. A concise and deep approach. See also the author's homepage (http://www.di.uminho.pt/~jno/html/) and also his many other papers (http://www.di.uminho.pt/~jno/html/jnopub.html) -- many materials related to this topic can be found there.