Making the Point-free Calculus Less Pointless

Alcino Cunha
Jorge Sousa Pinto

Departamento de Informática, Universidade do Minho
4710-057 Braga, Portugal
{alcino,jsp}@di.uminho.pt

APPSEM'04, April 16th

Introduction

Associated with the functional style of programming is an algebra of programs [. . .] This algebra can be used to transform programs and to solve equations whose "unknowns" are programs in much the same way one transforms equations in high-school algebra.

John Backus

- The main features of Backus functional style were the absence of variables and the use of specific *combinators* to combine existing functions into new functions.

- The choice of the combinators was based not only on their programming power, but also on the power of the associated algebraic laws.

- This style of programming is usually called *point-free*.

Introduction

- Although the point-free style has a rich calculus for reasoning about programs, most programmers resort to the point-wise style both for programming and for calculation.

- It is claimed that the point-free style is not very natural since the intuitive meaning of programs can easily be lost, and has been jokingly called the *pointless* style.

- In fact, some point-free derivations are very long and tedious, namely when dealing with higher-order functions.

- We aim at extending the calculus with new useful operators and tools that help reducing the burden of proofs just to the creative parts.

- This work is being carried out in the context of the PURe project (Program Understanding and Re-engineering: Calculi and Applications).

Catamorphisms

- Point-free programming is usually complemented with extensive use of *recursion patterns*.

- The best known is the *fold* or *catamorphism* - given a function of type $g : F\ A \rightarrow A$ is denoted by $(\!|g|\!)_F : \mu F \rightarrow A$, the function that builds its result by replacing the constructors of the input by g.

- One of the most important laws about this recursion operator is fusion.

$$f \circ (\!|g|\!)_F = (\!|h|\!)_F \quad \Leftarrow \quad f \circ g = h \circ F f \ \wedge \ f \text{ strict}$$

- For lists in Haskell we have the `foldr`.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

An Example

- Take the typical definition of reverse.

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = (reverse xs) ++ [x]
```

- An efficient version using accumulating parameters can be derived from the following specification using fusion and the associativity of concatenation.

```
reverse' :: [a] -> [a] -> [a]
reverse' l m = reverse l ++ m
```

- But first we need to state these equations using the point-free style and recursion patterns.

$$\mathsf{reverse}' = \overline{\mathsf{cat}} \circ (\!|\underline{\mathsf{nil}} \triangledown \mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})|\!)_{\underline{1} + \underline{A} \times \mathsf{Id}}$$

Point-free Uncurried Combinators

- The associativity property of cat is usually expressed as follows.

$$\mathsf{cat} \circ (\mathsf{id} \times \mathsf{cat}) \circ \mathsf{assocr} = \mathsf{cat} \circ (\mathsf{cat} \times \mathsf{id})$$

- A simpler calculation can be obtained if an uncurried version of the composition combinator was available.

$$\begin{array}{lcl} \mathsf{comp} & : & (C^B \times B^A) \to C^A \\ \mathsf{comp} & = & \overline{\mathsf{ap} \circ (\mathsf{id} \times \mathsf{ap}) \circ \mathsf{assocr}} \end{array}$$

- Associativity could then be more conveniently expressed as follows.

$$\overline{\mathsf{cat}} \circ \mathsf{cat} = \mathsf{comp} \circ (\overline{\mathsf{cat}} \times \overline{\mathsf{cat}})$$

Back to the Example

- From the calculation we get the following catamorphism.

$$\begin{array}{lcl} \mathsf{reverse}' & : & \mathsf{List}\ A \rightarrow (\mathsf{List}\ A \rightarrow \mathsf{List}\ A) \\ \mathsf{reverse}' & = & (\!|\underline{\mathsf{id}} \triangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{cons}} \times \mathsf{id})|\!)_{\underline{1}+\underline{A}\times \mathsf{Id}} \end{array}$$

- Indeed, even with for this simple example it is already difficult to understand its behavior. Lets move to back to the point-wise style.

```
reverse' :: [a] -> [a] -> [a]
reverse' = foldr (\x y z -> y (x:z)) id
```

- And finally introduce explicit recursion.

```
reverse' :: [a] -> [a] -> [a]
reverse' [] y     = y
reverse' (x:xs) y = reverse' xs (x:y)
```

A Left-Exponentiation Combinator

- Further complications arise if we want to apply the accumulation technique to derive functions with two accumulating parameters. For example, a tail recursive function to compute the height of a leaf tree can be derived from the following specification.

$$\begin{array}{lcl} \mathsf{height} & : & \mathsf{LTree}\ A \to (\mathsf{Nat} \to (\mathsf{Nat} \to \mathsf{Nat})) \\ \mathsf{height} & = & \overline{\mathsf{max}}^{\bullet} \circ \overline{\mathsf{plus}} \circ (\!|\underline{\mathsf{zero}} \triangledown \mathsf{succ} \circ \mathsf{max}|\!)_{\underline{A}+\mathsf{Id}\times\mathsf{Id}} \end{array}$$

- This calculation can be simplified by introducing the following left-exponentiation operator, for $f : B \to C$.

$$\begin{array}{lcl} {}^{\bullet}f & : & A^C \to A^B \\ {}^{\bullet}f & = & \overline{\mathsf{ap} \circ (\mathsf{id} \times f)} \end{array}$$

- The resulting tail-recursive height is $(\!|\underline{\overline{\mathsf{max}}} \triangledown {}^{\bullet}\mathsf{succ} \circ \mathsf{comp}^{\bullet} \circ \mathsf{split}|\!)$.

Ongoing and Future Work

- Pointless Haskell
  - Enables programming in Haskell in a true point-free style.
  - It includes a limited form of implicit coercion that allows types to be viewed as simple sums of products.

- Dr. Hylo
  - A tool that derives *hylomorphisms* from explicit recursive definitions.
  - It is being improved in order to derive point-free definitions and to identify more specific recursion patterns.

- Point-free Theorem Prover
  - Simple prototype based on a standard term rewriting and unification engine.
  - To what extent can proofs be fully automated?

Conclusions

- We believe that, with the appropriate machinery, the point-free style is indeed better for proving properties about functional programs.
- But we agree that programming in this style is not always recommended.
- We are developing tools that allow one to program in one style and calculate in the other.
- A useful comparison is that of mathematical transforms, such as the Fourier transform or the Laplace transform - the domain of definition is changed in order to make certain manipulations more easier to perform.
- For more information visit the PURe project website.

`http://www.di.uminho.pt/pure`