Is Point-free Pointless?

Alcino Cunha

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

PURe Workshop'04, September 13

Introduction

- Algebraic programming = point-free + recursion patterns.
- For a long time, we were told that this style is better for program calculation.
- There exists practical evidence that recursion patterns are useful for program transformation (HYLO, MAG, . . .). But what about point-free?
- Which of this components do we really treasure?
- What is the future of our research in algebraic programming?

Status of the Point-free Research at UMinho

- Theory
  - Reasoning with higher-order functions.
  - Converting point-wise to point-free.
  - Generalizing hylo-shift, inwards fusion, . . .

- Practice
  - Pointless: a library to program in the point-free style with recursion patterns.
  - DrHylo: a tool to derive hylomorphisms from recursive equations. Soon it will also be able to derive point-free definitions.
  - A tool to rewrite point-free definitions should also be on the way.

The Point-free Style

- How easy is to program with point-free?

- Consider Tree $A = \mu(1! \mathbin{\hat{+}} A! \mathbin{\hat{\times}} (\mathsf{Id} \mathbin{\hat{\times}} \mathsf{Id}))$. Lets write some traversals! Preorder is easy

$$\begin{array}{lcl} \mathsf{preorder} & : & \mathsf{Tree}\ A \rightarrow \mathsf{List}\ A \\ \mathsf{preorder} & = & (\!|\mathsf{nil} \mathbin{\nabla} \mathsf{cons} \circ (\mathsf{id} \times \mathsf{cat})|\!)_{\mathsf{Tree}\ A} \end{array}$$

  but inorder . . .

$$\begin{array}{lcl} \mathsf{inorder} & : & \mathsf{Tree}\ A \rightarrow \mathsf{List}\ A \\ \mathsf{inorder} & = & (\!|\mathsf{nil} \mathbin{\nabla} \mathsf{cat} \circ (\mathsf{id} \times \mathsf{cons}) \circ \mathsf{assocr} \circ (\mathsf{swap} + \mathsf{id}) \circ \mathsf{assocl}|\!)_{\mathsf{Tree}\ A} \end{array}$$

- Rearranging parameters is sometimes a nightmare . . .

The Point-free Style

- Another example: writing the inverse of $(A{\times}C){+}(B{\times}C) \to (A{+}B){\times}C$.

$$\mathsf{distl} = \mathsf{ap} \circ ((\overline{\mathsf{inl}} \mathbin{\triangledown} \overline{\mathsf{inr}}) \times \mathsf{id})$$

- Proving that $\mathsf{distl} \circ \mathsf{undistl} = \mathsf{id}$ is not difficult, but the point-wise proof is trivial - substitution is very powerful.

$$\begin{array}{ll}
 & \mathsf{ap} \circ ((\overline{\mathsf{inl}} \mathbin{\triangledown} \overline{\mathsf{inr}}) \times \mathsf{id}) \circ ((\mathsf{inl} \circ \mathsf{fst} \mathbin{\vartriangle} \mathsf{snd}) \mathbin{\triangledown} (\mathsf{inr} \circ \mathsf{fst} \mathbin{\vartriangle} \mathsf{snd})) \\
= & \quad \{\, \text{abides, product-absor} \,\} \\
 & \mathsf{ap} \circ ((\overline{\mathsf{inl}} \mathbin{\triangledown} \overline{\mathsf{inr}}) \circ (\mathsf{inl} \circ \mathsf{fst} \mathbin{\triangledown} \mathsf{inr} \circ \mathsf{fst}) \mathbin{\vartriangle} (\mathsf{snd} \mathbin{\triangledown} \mathsf{snd})) \\
= & \quad \{\, \text{sum-fusion, sum-strict, sum-cancel} \,\} \\
 & \mathsf{ap} \circ ((\overline{\mathsf{inl}} \circ \mathsf{fst} \mathbin{\triangledown} \overline{\mathsf{inr}} \circ \mathsf{fst}) \mathbin{\vartriangle} (\mathsf{snd} \mathbin{\triangledown} \mathsf{snd})) \\
= & \quad \{\, \text{abides, product-def} \,\} \\
 & \mathsf{ap} \circ ((\overline{\mathsf{inl}} \times \mathsf{id}) \mathbin{\triangledown} (\overline{\mathsf{inr}} \times \mathsf{id})) \\
= & \quad \{\, \text{sum-fusion, ap strict, exponentiation-cancel} \,\} \\
 & \mathsf{inl} \mathbin{\triangledown} \mathsf{inr} \\
= & \quad \{\, \text{product-reflex} \,\} \\
 & \mathsf{id}
\end{array}$$

The Point-free Style

- Could point-free be a good framework to implement mechanical reasoning and program transformation?

- Rewriting is very easy to implement, but even without recursion, neither laziness, there is no decidable rewriting system for equality in point-free.

- However, there are decision procedures for traditional lambda-calculus with unit, products, and sums.

- Given this, what is the point of converting from point-wise into point-free? Program understanding? distl is not that easy to understand. We have lots of “bad” examples with higher-order functions.

Reasoning About Haskell

- Is it correct to reason about Haskell using this framework?

- Consider cata uniqueness. What does it says about non-strict solutions?

$$f = (\!|g|\!)_{\mu F} \wedge g \text{ strict} \quad \Leftrightarrow \quad f \circ \mathsf{in}_{\mu F} = g \circ F f \wedge f \text{ strict}$$

- Every data type in Haskell is lifted:

$$(\bot, \bot) \neq \bot$$
$$\lambda x . \bot \neq \bot$$

- Should we use a different theory? One without categorical products, sums, and functions. Does it pays to continue in the categorical framework?

The Future

- Continue to ignore these problems or develop/move to a new theoretical framework? Something like P-logic? Or forget Haskell?

- Visual programming language for point-free - is it useful? It could help with some of the problems, like rearranging parameters.

- Should we move into a mixed style - point-wise and point-free?

- What about integrating some of our tools into the Haskell Refactorer? HaRe tools are also built on top of Programatica.

- Montevideo is also developing some tools (monadic fusion). How can we cooperate?