

Refactoring for Comprehension

Gustavo Villavicencio

Facultad de Matemática Aplicada
Universidad Católica de Santiago del Estero
Campus de la UCSE, 4200 Santiago del Estero, Argentina
gustavov@ucse.edu.ar

Abstract

Functional programming is well suited for equational reasoning on programs. In this paper, we are trying to use this capability for program comprehension purposes. Specifically, in a program understanding process, higher-order operators can work like abstract schemes in which we can fit formal specifications calculated from the source code. Such specifications are calculated by a transformational process which we call reverse program calculation that operates on both notations: pointwise and pointfree. Once a specification matches an abstract schema, a new refactoring phase leading to a clearer source code takes place. At the same time, an unambiguous behavioural understanding is reached because we give a mathematical description of the abstract schemes. To provide a more complete and realistic perspective of the approach, we use recursive operators that can handle side effects.

Keywords: Refactoring, program understanding, category theory, algebra of programming, monads, reverse program calculation.

1 INTRODUCTION

Reverse engineering or program comprehension (the differences are irrelevant for our purpose) is the process by which usually starting from the source code, one can create representations of a software system at a higher level of abstraction [2]. This process is activated when maintenance activities are performed on poorly documented systems or on systems whose design has been lost through years of maintenance.

In previous works [20, 24] we have shown how to use functional support for reasoning on imperative programs by reverse calculation. That is, starting from imperative source code we obtain the corresponding slices on a functional setting, and for each of them, we calculate a specifications support expressed as a set of equations in *pointfree style* [4, 10, 21].

Following the same way of program understanding on the functional setting, Gibbons [11] shows how to use the reverse of *fusion*, i.e. a *fission* process, to recover the lost design.

In this paper, we investigate a complementary view of the process outlined in [20, 24], exploiting the support of the functional programming paradigm to handle higher-order operators. Precisely, these operators can work like abstract schemes where the calculated specifications fit. That is, given a code fragment, the reverse

program calculation process obtains its specification represented in terms of an abstract scheme. By formal manipulation of the specifications we can get a refactored version of the original code fragment.

From the point of view of [11], we might see the refactoring process as a mechanism to recover the lost design. From this perspective, monadic operators can be viewed as higher-order recursion patterns by means of which we “document” the recovered architecture. So, the process in which we are interested might be viewed as a program comprehension process which tries to fit the recovered specification (calculated for us) into a formal pattern.

We have organized the paper as follows. First, an overview on monads and the **State** monad specifically, is presented in section 2. Section 3 describes the approach we are proposing and the abstract schemes with which we have experimented until now. Then, an example is developed in section 4. Next, we show the directions to follow and the conclusions in section 5, and finally the acknowledgements.

2 MONADS

Formally, a monad [19] is a triple $T = (T, \eta, \mu)$ in a category C is a (endo)functor $T : C \rightarrow C$ and two *natural transformations* (polymorphic functions) $\eta : id_C \Rightarrow T$ and $\mu : TT \Rightarrow T$ which satisfy the following properties: $\mu \cdot \mu T = \mu \cdot T\mu$, $\mu \cdot T\eta = \mu \cdot \eta T = id_C$.

In the functional programming setting, monads are interpreted as a *Kleisli triple*. A Kleisli triple $(M, \eta, -^*)$ in a category C is composed by a functor at objects level $M : Obj(C) \rightarrow Obj(C)$, a natural transformation $\eta : I \Rightarrow M$, and an *extension operator* $f^* : MA \rightarrow MB$ where $f : A \rightarrow MB$, such that the following properties are fulfilled: $\eta_A^* = id_{MA}$, $f^* \cdot \eta_A = f$, and $f^* \cdot g^* = (f^* \cdot g)^*$, where $f : A \rightarrow MB$ and $g : B \rightarrow MC$. The extension operator supplies a mechanism to compose monadic functions. i.e. *Kleisli composition*: $g \bullet f \stackrel{def}{=} g^* \cdot f$. Based on this definition, we can determine that the two first laws previously indicated, state that η is the right and left identity, and the last one establishes that the composition is associative.

Given a Kleisli triple $(M, \eta, -^*)$, the *Kleisli category* C_M is defined as follows: The **objects** in C_M are the same as in C , the set of **morphisms** $C_M(A, B) \equiv C(A, MB)$, **identity** is given by $\eta_A : A \rightarrow MA$, and **composition** by Kleisli composition.

Specifically, in the functional programming setting the kleisli triple can be expressed in the following way: $(M, unit, \star)$, where M is a type constructor, $unit : A \rightarrow MA$ a polymorphic function, and $\star : MA \times (A \rightarrow MA) \rightarrow MA$ a polymorphic operator, usually called *bind*. So, the monadic composition in functional programming is denoted by $m \star f$ which, in kleisli notation, would be $f^*(m)$. That is, m is evaluated generating a result that is taken by f as input, and then f is evaluated. Using lambda notation we would have: $m \star \lambda v. f v$.

We can define functors $(\widehat{\quad}) : C \longrightarrow C_M$ and vice-versa $U : C_M \longrightarrow C$. For our purposes, it is particularly interesting the first one which is named as a *lifting functor* and defined as the identity for objects and $\widehat{f} = \text{unit}_B \cdot f : A \longrightarrow MB$ for morphisms $f : A \longrightarrow B$.

Because the example developed in section 4 requires the **State** monad, in the following sections we introduce a quick overview on this topic.

2.1 Exponentials

Exponential datatypes are used to model the **State** monad. It is defined as $B^A \stackrel{\text{def}}{=} \{g \mid g : A \longrightarrow B\}$, which means that B^A is an alternative notation for $g : A \longrightarrow B$. Therefore, we can write expressions as $g \in B^A$ which at the same time, we lead to construct operators that accept functions as argument and apply them

$$\begin{aligned} ap & : B^A \times A \longrightarrow B \\ ap(g, a) & = g a \end{aligned} \quad (1)$$

By other hand, in functions as $f : C \times A \longrightarrow B$ the C argument can be “frozen” so that $f_c : A \longrightarrow B$, i.e. $f_c \in B^A$. But this kind of functions can be regarded with signature $C \longrightarrow B^A$, and could be denoted as

$$\begin{aligned} \bar{f} & : C \longrightarrow B^A \\ (\bar{f}c)a & = f(c, a) \end{aligned} \quad (2)$$

In some literature these functions are known as “transposes” [21] but they are better known in the functional setting as *curry functions*.

In the same way as $A \times B$ (product) and $A + B$ (coproduct), B^A (exponential) has useful properties: cancellation, reflexion, fusion, absorption, exponential-functor, and exponentials-functor-id. Among these, will be particularly useful during the example, the exponentials fusion property

$$\overline{g \cdot (f \times id)} = \bar{g} \cdot f \quad (3)$$

The other properties mentioned as much product and coproduct as exponential will not be developed here, and the reader will have to consult the appropriate literature [9, 18] et al.

2.2 The State Monad

In imperative programming the state is the collection of all global variables and it is passed from one function to another in a sequential way. Therefore, if we have function $f : A \longrightarrow B$ we can use the next expression to model the implicit state passing in the imperative context as $f \times id_S : A \times S \longrightarrow B \times S$.

On the other hand, as usual, the monad is represented algebraically as a triple $\mathbf{S} = (M, \eta, \mu)$ where

- For an object A , $MA = (A \times S)^S$

- For a morphism $f : A \longrightarrow B$, $Mf = \overline{(f \times id_S) \cdot app_{S,A \times S}} : (A \times S)^S \longrightarrow (B \times S)^S$
- For an object A , $\eta_A = \overline{id_{A \times S}} : A \longrightarrow (A \times S)^S$
- For an object A , $\mu_A = \overline{app_{S,A \times S} \cdot app_{S,SA \times S}} : ((A \times S)^S \times S)^S \longrightarrow (A \times S)^S$

We must also note, that monadic functions $f : A \longrightarrow (B \times S)^S$ in \mathcal{C} can be obtained from $f : A \longrightarrow B$ in \mathcal{C}_M by $f \times id$. Further, $Mf = (f \times id)^*$ property that we leave to the reader to prove.

3 THE REVERSE PROGRAM CALCULATION PROCESS

To make the understanding of the reverse program calculation process easy, it is usually compared with the *Laplace transformation*. Based on this contrast, we construct a schematic view shown in figure 1.

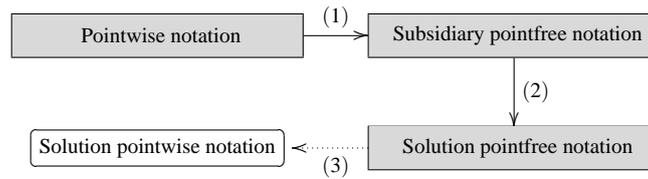


FIGURE 1. Laplace approach applied to RPC process

Starting from pointwise expressions [5], usually written in **HASKELL** [14] or **VDM-SL** [12], we try to obtain an equivalent pointfree expression. To reach this target, successive transformation steps are performed on the pointwise expressions. The process finishes when we obtain an expression that is handled by some property or law accessible from a calculus. Up to now, these source-to-source transformations are intuitive, and some kind of rules are required to direct this process.

Commonly, to clarify which is the law or property involved, a commutative diagram is constructed from the last pointwise expression obtained. Why are we translating pointwise expressions into pointfree expressions? Because the reasoning process is easier in the pointfree side, and because this kind of notation is more compact and abstract than pointwise notation [10, 4]. This is the process represented by arrow (1) in figure 1.

Once on the pointfree side, the real calculational process is performed. From the commutative diagram constructed in the previous phase, we extract a pointfree expression which is subject to the application of laws and properties accessible from the calculus. The process ends when a solution is calculated, i.e. when no

law and property can be applied. This is the process sketched by arrow (2) in figure 1.

At this point, we have calculated a very compact and abstract program representation, and therefore, a formal reverse engineering [2] process has been outlined. We call this process *Reverse Program Calculation* (RPC), and the abstract program representation is really an equation that “describes” precisely the program behaviour. The filled boxes in figure 1 represent the RPC process.

But, just as the equation explains the program behaviour, it also allows us to articulate the RPC process with a refactoring process, making the whole process a formal reengineering [2] one (arrow (3) in figure 1). Specifically, we will see in the example that the calculated equation from the function we are interested in fits the formal definition of the corresponding monadic operator. In this way, an articulation between the RPC process and a refactoring process occurs.

From the perspective of [11], the RPC process + the refactoring process is a *program comprehension* process. In this view, the essence of a design is expressed in terms of higher-order recursion patterns, that is, monadic recursive operators in the context of this paper.

In our case, the refactoring phase generates a reimplementaion in the same language as the original source code, i.e. **HASKELL**. However, we can think in a more *generic* refactoring phase generating results like in **VDM-SL**, for instance.

Unfortunately, in the present state of research, the described process is completely manual, and therefore impracticable on real cases. The reader can appreciate some application difficulties during the development of an example in the next section. These will be discussed in more detail in section 5.

3.1 Monadic Recursive Operators

In this section, we will introduce the higher order operators that work like abstract schemes in our approach and with which we have experimented up to date: monadic catamorphism, monadic anamorphism and monadic hylomorphism.

Nevertheless, before introducing such concepts, a quick overview to the concept of *monadic extension of a functor* is necessary. It is a construction of type $\widehat{F} : C_M \longrightarrow C_M$ such that $\widehat{F}A = FA$, and on monadic morphisms $f : A \longrightarrow MB$, it yields $\widehat{F}f : \widehat{F}A \longrightarrow M(\widehat{F}B)$ in C_M , or what is the same $\widehat{F}f : FA \longrightarrow M(FB)$.

The construction of \widehat{F} requires the application of a natural transformation $\delta^F : FM \Rightarrow MF$ that distributes a monad over a functor. This natural transformation is called a *distribution law* [1, 22].

In order to show how the distribution law for the list functor is calculated, which will be used afterward, we introduce the concept of *strength*. It is a natural transformation τ induced by the *strengths* with which strong functors are equipped, and is defined as follows

$$\begin{aligned} \tau_{A,B} &:: A \times MB \longrightarrow M(A \times B) \\ \tau_{A,B}(a, m) &= m \star \lambda b. \text{unit}(a, b) \end{aligned} \quad (4)$$

The distribution law for the sum functor is $\delta_{A,B}^+ = [Mi_1, Mi_2]$ (This definition can be used with some restrictions in **Cpo** since in this category there is not coproduct), and from here we can calculate the distribution law for the functor $L_A = \underline{1} + \underline{A} \times I$ (where $\underline{\cdot}$ denotes the constant functor and I the identity functor), that is, the functor that captures the signature for lists. We proceed as follows

$$\begin{aligned}
& [Mi_1, Mi_2] \cdot (unit + \tau_{(A,L)}) \\
= & \{ \text{action of monad } M \text{ in } \mathcal{C}_M \} \\
& [(unit \cdot i_1)^*, (unit \cdot i_2)^*] \cdot (unit + \tau_{A,L}) \\
= & \{ \text{lifting functor} \} \\
& [\widehat{i}_1^*, \widehat{i}_2^*] \cdot (unit + \tau_{A,L}) \\
= & \{ +\text{-absortion} \} \\
& [\widehat{i}_1^* \cdot unit, \widehat{i}_2^* \cdot \tau_{A,L}] \\
= & \{ \text{second kleisli triple property and kleisli composition} \} \\
& [\widehat{i}_1, \widehat{i}_2 \bullet \tau_{A,L}]
\end{aligned}$$

therefore

$$\begin{aligned}
\delta^{L_A} & : \quad 1 + A \times MX \longrightarrow M(1 + A \times X) \\
\delta^{L_A} & = \quad [\widehat{i}_1, \widehat{i}_2 \bullet \tau_{A,X}]
\end{aligned} \tag{5}$$

3.1.1 Monadic Catamorphism

A *monadic fold* (or monadic catamorphism) is a function that behaves like a fold, but with the additional feature of producing effects. As an approximation to its definition we consider the next commutative diagram in the Kleisli category \mathcal{C}_M

$$\begin{array}{ccc}
MA & \xleftarrow{h} & FA & f \bullet h = h' \bullet \widehat{F}f \\
f^* \downarrow & & \downarrow \widehat{F}f & \\
MB & \xleftarrow{h'} & MFB &
\end{array} \tag{6}$$

In this view, we are thinking of functions that involve a recursive process during which side effects can be produced. From the previous diagram we can infer the property as we see above on the right, where $h : FA \rightarrow MA$ and $h' : FB \rightarrow MB$ are monadic algebras and $f : A \rightarrow MB$ a homomorphism between them.

By definition, and supposing that $(M\mu F, \widehat{in}_F)$ is the initial monadic algebra, then there is a unique homomorphism to any monadic algebra $f : B \rightarrow MB$. In a diagram

$$\begin{array}{ccc}
M\mu F & \xleftarrow{\widehat{in}_F} & F\mu F \\
\langle f \rangle \downarrow & & \downarrow \widehat{F}\langle f^* \rangle \\
M B & \xleftarrow{f^*} & M F B
\end{array} \quad h = \langle f \rangle \iff h \bullet \widehat{in}_F = f \bullet \widehat{F}h \quad (7)$$

So, the *monadic fold* operator [6], $\langle f \rangle_F^M : \mu F \longrightarrow M B$ is then defined as the least homomorphism between \widehat{in} and f [22].

Using the fact that $h \bullet \widehat{in}_F = h \cdot in_F$ and $\widehat{F}h = \delta_B^F \cdot F h$, we can rewrite (7) as

$$\begin{array}{ccc}
\mu F & \xleftarrow{in} & F\mu F \\
\langle f \rangle \downarrow & & \downarrow F\langle f \rangle \\
M B & \xleftarrow{f^*} M F B & \xleftarrow{\delta_B^F} F M B
\end{array} \quad h \cdot in_F = (f \bullet \delta_B^F) \cdot F h \quad (8)$$

In this way, every homomorphism $f : \mu F \longrightarrow M B$ between the monadic algebras \widehat{in}_F and f , is also a homomorphism between the normal algebras in_F and $f \bullet \delta_B^F : F M B \longrightarrow M B$, and vice-versa [22].

3.1.2 Monadic Anamorphism

As in the case of catamorphism, we can define an *anamorphism* as the least homomorphism, but now, between the *monadic coalgebras* $h : A \longrightarrow M(FA)$ and \widehat{out}_F

$$\begin{array}{ccc}
M\mu F & \xrightarrow{\widehat{out}_F^*} & M(F\mu F) \\
\uparrow [(h)_F^M] & & \uparrow (\widehat{F}[(h)_F^M])^* \\
A & \xrightarrow{h} & M(FA)
\end{array} \quad \widehat{out}_F \bullet [(h)_F^M] = \widehat{F}[(h)_F^M] \bullet h \quad (9)$$

and because \widehat{in}_F and \widehat{out}_F are isomorphisms and one the inverse of the other, we can rewrite (9) as

$$\begin{array}{ccc}
M\mu F & \xleftarrow{\widehat{in}_F^*} & M(F\mu F) \\
\uparrow [(h)_F^M] & & \uparrow (\widehat{F}[(h)_F^M])^* \\
A & \xrightarrow{h} & M(FA)
\end{array} \quad [(h)_F^M] = \widehat{in}_F \bullet \widehat{F}[(h)_F^M] \bullet h \quad (10)$$

Although monadic anamorphism will not be used in the example, it will be required for the definition of *monadic hylomorphism* in the next section, which will be useful in the example.

3.1.3 Monadic Hylomorphism

Intuitively, this operator represents functions that generate effects both during, construction and consumption of the intermediate data structure.

As with normal algebras, we would expect to avoid the generation of the intermediate data structure, performing some transformations to obtain the next definition

$$\begin{array}{ccc}
 MA & \xleftarrow{h^*} & M(FA) \\
 \uparrow (\langle h \rangle_F^M) \bullet [(g)]_F^M & & \uparrow (\widehat{F}((\langle h \rangle_F^M) \bullet [(g)]_F^M))^* \\
 B & \xrightarrow{g} & M(FB)
 \end{array}
 \qquad
 \llbracket h, g \rrbracket_F^M = h \bullet \widehat{F}(\langle h \rangle_F^M \bullet [(g)]_F^M) \bullet g$$

(11)

We can prove that (8) is an instance of (11) making $g = \widehat{out}_F$, and also that, (10) is an instance of (11) making $h = \widehat{in}_F$.

4 AN EXAMPLE

In figure 2, we show a **HASKELL** program that sums a sequence of numbers, and returns a pair (result, length of the sequence). Function *tick* increments the state every time a computation is performed. *get* and *put* are methods of the *MonadState* class. That is, we are using the monad **State** to know the length of the sequence of numbers.

```

tick :: State Int Int
tick = do c <- get
         put (c+1)
         return c

sms :: [Int] -> Int -> State Int Int
sms [] = \s -> return 0
sms (e:l) = \s -> do tick
                    r <- sms l s
                    return (r+e)

evalsms :: [Int] -> Int -> (Int,Int)
evalsms l = \s -> runState (sms l s) s

```

FIGURE 2. Program example using the State monad

We have selected a simple example, since we consider it is enough to show the ideas in we are interested in. Although the program is simple, we will see that the RPC process performed below is not (unfortunately). From figure 2, we want to construct a commutative diagram as shown in (12).

$$\begin{array}{ccc}
L & \xleftarrow{\quad in \quad} & 1 + Int \times L \\
\langle sms \rangle \downarrow & & \downarrow id + id \times \langle sms \rangle \\
(Int \times S)^S & \xleftarrow[\langle h \times id_S \rangle^*]{} ((1 + Int \times Int) \times S)^S \xleftarrow[\delta_{Int}^L]{} 1 + Int \times (Int \times S)^S &
\end{array} \tag{12}$$

On the other hand, it is important to highlight that in real conditions, source-to-source transformations would be necessary before obtaining a pointwise expression that might be handled by some law or property accessible from the calculus.

From the previous commutative diagram we can extract the next property

$$\begin{aligned}
& \langle sms \rangle \cdot in \\
= & \{ \text{commutative diagram} \} \\
& \overline{h \times id} \bullet \delta_{Int}^L \cdot (id + id \times \langle sms \rangle) \\
= & \{ \text{distribution law definition} \} \\
& \overline{h \times id} \bullet [\widehat{i}_1, \widehat{i}_2 \bullet \tau_{Int,L}] \cdot (id + id \times \langle sms \rangle) \\
= & \{ \text{kleisli composition definition} \} \\
& \overline{(h \times id)^*} \cdot [\widehat{i}_1, \widehat{i}_2 \bullet \tau_{Int,L}] \cdot (id + id \times \langle sms \rangle) \\
= & \{ \text{+-fusion} \} \\
& \overline{(h \times id)^* \cdot \widehat{i}_1, (h \times id)^* \cdot \widehat{i}_2 \bullet \tau_{Int,L}} \cdot \\
& (id + id \times \langle sms \rangle) \\
= & \{ \text{lifting functor definition} \} \\
& \overline{(h \times id)^* \cdot (unit \cdot i_1), (h \times id)^* \cdot (unit \cdot i_2) \bullet \tau_{Int,L}} \cdot \\
& (id + id \times \langle sms \rangle) \\
= & \{ \text{associativity and second kleisli triple property} \} \\
& \overline{(h \times id) \cdot i_1, (h \times id) \cdot i_2 \bullet \tau_{Int,L}} \cdot \\
& (id + id \times \langle sms \rangle) \\
= & \{ \text{exponential fusion in reverse} \} \\
& \overline{(h \times id) \cdot (i_1 \times id), (h \times id) \cdot (i_2 \times id) \bullet \tau_{Int,L}} \cdot \\
& (id + id \times \langle sms \rangle) \\
= & \{ \text{“bi-distribution” of } \times \text{ with respect to composition in reverse} \} \\
& \overline{(h \cdot i_1) \times (id \cdot id), (h \cdot i_2) \times (id \cdot id) \bullet \tau_{Int,L}} \cdot \\
& (id + id \times \langle sms \rangle) \\
= & \{ \text{identity and } h \text{ definition} \} \\
& \overline{([\mathbb{0}, +] \cdot i_1) \times id, ([\mathbb{0}, +] \cdot i_2) \times id \bullet \tau_{Int,L}} \cdot \\
& (id + id \times \langle sms \rangle) \\
= & \{ \text{+-cancellation} \}
\end{aligned}$$

$$\begin{aligned}
& \overline{[0 \times id, (+ \times id)]} \bullet \tau_{Int,L} \cdot (id + id \times \langle \langle sms \rangle \rangle) \\
= & \{+-absortion \text{ and kleisli composition definition}\} \\
& \overline{[0 \times id, (+ \times id)]}^* \cdot \tau_{Int,L} \cdot (id \times \langle \langle sms \rangle \rangle)
\end{aligned}$$

Since $in = [Nil, Cons]$ we can conclude that

$$\begin{aligned}
\langle \langle sms \rangle \rangle Nil &= \overline{0 \times id_S} \\
\langle \langle sms \rangle \rangle (Cons) &= \overline{(+ \times id_S)}^* \cdot \tau_{Int,L} \cdot (id \times \langle \langle sms \rangle \rangle)
\end{aligned} \tag{13}$$

So, we have calculated a solution, i.e. an equation that expresses, without ambiguity, the behaviour of the fragment that is being analysed. At this point, we have performed the RPC process outlined in section 3, i.e. arrows **(1)** and **(2)**. In other words, we have executed a formal reverse engineering process: starting from **HASKELL** expressions, we have calculated a more abstract specification.

In the next section, we will go through arrow **(3)** to observe how the previously calculated equation is useful to refactor the original function in **HASKELL**.

4.1 Monadic Refactoring

Many monadic recursive operators that encapsulate recursive schemes of programs that produce effects have been defined [22, 23]. In this section, we are going to show how (13) can be rewritten in terms of one of the monadic recursive operators, namely monadic fold. In our example, the recursive scheme involved is the well known monadic fold operator on lists. Therefore, in order to calculate it from the monadic fold definition [22] we proceed as follows:

$$\begin{aligned}
& \langle \langle h \rangle \rangle \cdot in_F \\
= & \{\text{definition}\} \\
& (h \bullet \delta_A^f) \cdot F \langle \langle h \rangle \rangle \\
= & \{\text{list distribution law and list functor}\} \\
& (h \bullet [\widehat{i}_1, \widehat{i}_2 \bullet \tau_{A,L}]) \cdot (id + id \times \langle \langle h \rangle \rangle) \\
= & \{\text{associativity}\} \\
& h \bullet ([\widehat{i}_1, \widehat{i}_2 \bullet \tau_{A,L}] \cdot (id + id \times \langle \langle h \rangle \rangle)) \\
= & \{+-absortion\} \\
& h \bullet [\widehat{i}_1, \widehat{i}_2 \bullet \tau_{A,L}] \cdot (id \times \langle \langle h \rangle \rangle) \\
= & \{+-fusion\} \\
& [h \bullet \widehat{i}_1, h \bullet \widehat{i}_2 \bullet \tau_{A,L}] \cdot (id \times \langle \langle h \rangle \rangle) \\
= & \{h \text{ definition and +-fusion}\} \\
& [[h_1, h_2] \bullet \widehat{i}_1, [h_1, h_2] \bullet \widehat{i}_2 \bullet \tau_{A,L}] \cdot (id \times \langle \langle h \rangle \rangle) \\
= & \{\text{kleisli composition}\} \\
& [[h_1, h_2]^* \bullet \widehat{i}_1, [h_1, h_2]^* \bullet \widehat{i}_2 \bullet \tau_{A,L}] \cdot (id \times \langle \langle h \rangle \rangle)
\end{aligned}$$

$$\begin{aligned}
&= \{\text{lifting}\} \\
&\quad [[h_1, h_2]^* \cdot (\text{unit} \cdot i_1), [h_1, h_2]^* \cdot (\text{unit} \cdot i_2) \bullet \\
&\quad \tau_{A,L} \cdot (\text{id} \times \langle h \rangle)] \\
&= \{\text{associativity}\} \\
&\quad [([h_1, h_2]^* \cdot \text{unit}) \cdot i_1, ([h_1, h_2]^* \cdot \text{unit}) \cdot i_2 \bullet \\
&\quad \tau_{A,L} \cdot (\text{id} \times \langle h \rangle)] \\
&= \{\text{second kleisli triple property}\} \\
&\quad [[h_1, h_2] \cdot i_1, [h_1, h_2] \cdot i_2 \bullet \tau_{A,L} \cdot (\text{id} \times \langle h \rangle)] \\
&= \{\text{+-cancellation}\} \\
&\quad [h_1, h_2 \bullet \tau_{A,L} \cdot (\text{id} \times \langle h \rangle)]
\end{aligned}$$

Therefore, we have that

$$\begin{aligned}
\langle h \rangle Nil &= h_1 \\
\langle h \rangle Cons &= h_2^* \cdot \tau_{A,L} \cdot (\text{id} \times \langle h \rangle)
\end{aligned} \tag{14}$$

where $\tau_{A,B}$ is the natural transformation defined by (4). Therefore, from (14) we can derive a version (in **HASKELL**, for example) of this operator as in figure 3.

```

mfoldL :: (Monad m) => (m b, a -> b -> m b) -> [a] -> m b
mfoldL (h1,h2) = mf
  where mf []     = h1
        mf (e:l) = do x <- mf l
                      h2 e x

```

FIGURE 3. Uncurried version of mfold operator for lists

Therefore, “matching” (13) and (14) we note that $h_1 = \overline{0 \times id_S}$ and $h_2 = \overline{+ \times id_S}$. So, we can rewrite (13) as shown in figure 4.

```

sms l = \s -> mfoldL(return 0, \x y -> do {c <- tick;
                                           return(x+y)}) l

```

FIGURE 4. sms function refactored by mfoldL operator

It is noted by [23], that the **mfoldL** operator performs the actions in a compulsory sequence. So, when an action is placed between the recursive calls, **mfoldL** fails to implement the function. See the example in [23] for details.

This problem is due to the application of the monadic extension \widehat{F} as a mechanism to structure recursive calls. To overcome this problem, we need an algebra with monadic carrier, i.e. with type $F(MA) \longrightarrow MA$.

To start drawing a solution, we rewrite the definition of monadic hylomorphism definition [22] in a clearer way in order to make the distribution law δ_A^F explicit

$$\begin{array}{ccc}
B & \xrightarrow{g} & M(FB) \\
\Downarrow \llbracket h \bullet \delta_A^F, g \rrbracket & & \Downarrow (F \llbracket h \bullet \delta_A^F, g \rrbracket)^* \\
MA & \xleftarrow{h^*} M(FA) \xleftarrow{\delta_A^F} & F(MA)
\end{array} \quad (15)$$

Thus, we can see that the algebra $h \bullet \delta_A^F$ has type $F(MA) \rightarrow MA$. In this way, if we try to disarticulate \widehat{F} we need a new algebra h' with that type, i.e. an algebra with monadic carrier.

With this modification, the problem to solve now is to preserve the kleisli composition since h' does not compose with it. We resort to the $mmap$ function in [23] to overcome this problem. Now, we are in condition to rewrite (15) as the commutative diagram shown in (16).

$$\begin{array}{ccc}
B & \xrightarrow{g} & M(FB) \\
\Downarrow \llbracket h', g \rrbracket & & \Downarrow M(F \llbracket h', g \rrbracket) \\
MA & \xleftarrow{h'^*} & M(F(MA))
\end{array} \quad \llbracket h', g \rrbracket = h' \bullet M(F \llbracket h', g \rrbracket) \bullet g \quad (16)$$

Obviously, we can prove that from this commutative diagram we can calculate a new definition of monadic catamorphism by taking $g = \widehat{out}_F$

$$\begin{aligned}
& \llbracket h', \widehat{out}_F \rrbracket \\
= & \quad \{ (16) \text{ and functor application} \} \\
& h' \bullet (unit \cdot F(\llbracket h', \widehat{out}_F \rrbracket))^* \cdot \widehat{out}_F \\
= & \quad \{ \text{kleisli composition and lifting} \} \\
& h' \bullet (unit \cdot F(\llbracket h', \widehat{out}_F \rrbracket)) \cdot out_F \\
= & \quad \{ \text{kleisli definition and associativity} \} \\
& (h'^* \cdot unit) \cdot F(\llbracket h', \widehat{out}_F \rrbracket) \cdot out_F \\
= & \quad \{ \text{second law in kleisli triple} \} \\
& h' \cdot F(\llbracket h', \widehat{out}_F \rrbracket) \cdot out_F
\end{aligned}$$

therefore, and because in_F and out_F are isomorphisms

$$\langle \llbracket h' \rrbracket \rangle \cdot in_F = h' \cdot F \langle \llbracket h' \rrbracket \rangle \quad (17)$$

which is the catamorphism definition between normal algebras, but here, h' is an algebra with monadic carrier. Starting from here, we can calculate the monadic catamorphism for lists, for example, obtaining the next equation

$$\begin{aligned} \langle h' \rangle Nil &= h'_1 \\ \langle h' \rangle Cons &= h'_2 \cdot (id \times \langle h' \rangle) \end{aligned} \quad (18)$$

which can be written in **HASKELL** as shown in figure 5

```
nmfoldL :: Monad m => (m a, m b -> m a -> m a) -> [b] -> m a
nmfoldL (h1,h2) = mfl
  where mfl []      = h1
        mfl (a:as) = h2 (return a) (mfl as)
```

FIGURE 5. mfold operator for lists without distribution law

After the calculational process, with the new operator we obtain $h'_1 = \overline{0} \times id_S$ and $h'_2 = \overline{+} \times id_S$, which leads to a new refactored version of *sms* based on **nmfoldL**, shown in figure 6

```
sms = \s -> nmfoldL(return 0, \e r -> do {c <- tick; x <- e;
                                          y <- r; return(x+y)})
```

FIGURE 6. sms refactored by nmfoldL operator

Therefore, we have calculated two alternative refactored versions of *sms* function. Obviously, cases where **mfoldL** fails to work, like that presented in [23], could be refactored by **nmfoldL**. Someone might argue that such refactored versions of the *sms* function in figures 4 and 6 are harder to understand than the original version.

However, the “formal explanation” behind them, equations (14) and (18) respectively, represent the kind of understanding we are interested in. In this way, the calculated pointfree specifications of the original code fragment fit a higher-order recursive pattern (monadic fold in our example) which explains its behaviour. Moreover, the refactored versions prove how the “lost structure” of the original program (as argued in [11]) has been recovered.

5 FUTURE WORKS AND CONCLUSIONS

As we have already said, the source-to-source transformations are not guided by any pre-existent rule or transformational schema. We argue also that they are far from being supported by automatic tools like **HaRe** [16]. However, there seems to be an alternative way to overcome this problem. Specifically, in the context of

the PURE Project (<http://lmf.di.uminho.pt/wiki/bin/view/PURE/WebHome>) some work [3] is carried out in order to translate a pointwise expression into its point-free equivalent. If this technology were available (monadic transformations are not covered by that work yet), the whole calculation process would be performed in the pointfree side.

The calculation process in the pointfree side also requires an automatic support. The developed example uses some few algebraic laws, but more complex cases will use a more important body of laws and properties. So, an automatic assistant for analysts would be necessary during real working conditions.

Regarding theoretical aspects, we have also focused on solving cases involving monad combinations [13, 15]. Specifically, we are trying to solve situations where we are using monad transformers [17], and studying alternative mechanisms to compose monads such as *co-products* [7]. A first challenge on these issues would be to construct abstract schemes, i.e. monadic recursive operators as we have applied here, but involving monad combinations according the different mechanisms of composition.

On the other hand, the experiments carried out so far involve only the computational patterns mentioned. Nevertheless, the approach might scale up to more abstract schemes, which can be calculated and reused. For example, [8] proposes the construction of *datatype-generic* patterns of computation by parametrizing the shape of the data they manipulate.

To abridge the conclusions:

1. We can calculate abstract schemes to work as patterns for program comprehension.
2. We can calculate specifications, i.e. mathematical equations, from a code fragment that fits the calculated patterns.
3. We can manipulate the patterns mathematically.

In this way, we emphasize the fact that functional context provides us with a strong setting for program comprehension and formal reengineering software systems.

On the other hand, we have carried out more experiments involving other datatypes and monads, and the obtained results support our argument that the current approach can scale up. The results reported by [3] regarding the automatic support, and also by [11], are in the same direction.

6 ACKNOWLEDGEMENTS

First, I would like to thank Dr. Alberto Pardo (Universidad de la República, Montevideo, Uruguay) for his invaluable comments and suggestions to improve this paper. Many thanks also to the reviewers for their appropriate observations on the abstract version of this document.

REFERENCES

- [1] M. Barr and Ch. Wells. *Toposes, Triples and Theories*. November 2002. Revised version - Version 1.1.
- [2] E.J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, 1990.
- [3] Alcino Cunha. Point-free programming with hylomorphisms. In *Workshop on Datatype-Generic Programming*, Oxford, England, June 2004. <http://web.comlab.ox.ac.uk/oucl/research/pdt/ap/dgp/workshop2004/>.
- [4] Alcino Cunha and Jorge Sousa Pinto. Making the point-free calculus less pointless. In *2nd. APPSEM II Workshop*, pages 178–179, 2004.
- [5] Oege de Moor and Jeremy Gibbons. Pointwise relational programming. In *AMAST: 8th International Conference on Algebraic Methodology and Software Technology*, volume 1816 of *LNCS*, pages 371–390. Springer Verlag, May 2000.
- [6] Maarten M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Technical Report Memoranda Inf 94-28, University of Twente, Enschede, Netherlands, June 1994.
- [7] Neil Ghani and Christoph L uth. Composing monads using coproducts. *Intl. Conference on Functional Programming 2002*, 37(9):133–144, 2002.
- [8] Jeremy Gibbons. Design patters as higher-order datatype generic programs. In *Workshop on Generic Programming*, Portland, Oregon, USA, September. ACM SIGPLAN.
- [9] Jeremy Gibbons. An introduction to the Bird-Meertens formalism. New Zeland Formal Program Development Colloquium Seminar, November 1994.
- [10] Jeremy Gibbons. A pointless derivation of radix sort. *Journal of Functional Programming*, 9(3):339–346, 1999.
- [11] Jeremy Gibbons. Fission for program comprehension. In *8th Int. Conference on Mathematics of Program Construction*, volume 4014 of *LNCS*. Springer Verlag, July 2006.
- [12] IFAD. Vdm tools. Technical report, IFAD, Forskerparken 10, DK-5230 Odensen M, Denmark, 1999. <http://www.ifad.dk>.
- [13] M. Jones and L. Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Dept. of Computer Science, Yale University, New Haven, Connecticut, USA, December 1993.
- [14] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries*. Cambridge University Press, Cambridge, UK, April 2003.
- [15] D. King and P. Wadler. Combining monads. In *Glasgow workshop on functional programming*, Workshops in Computing. Springer Verlag, July 1992.
- [16] Huiqing Li, Claus Reinke, and Simon Thompson. Tool support for refactoring functional programs. In Johan Jeuring, editor, *ACM SIGPLAN 2003 Haskell Workshop*. ACM, August 2003.
- [17] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, CA, January 1995. ACM Press, New York, 1995.
- [18] Lambert Meertens. Category theory for program construction by calculation. Lecture Notes for ESSLLI'95, 1995. Barcelona, Espa na.

- [19] E. Moggi. Notions of computations and monads. *Informations and Computations*, 93(1):55–92, 1991.
- [20] J. N. Oliveira and G. Villavicencio. Reverse program calculation supported by code slicing. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 35–45, Stuttgart, Germany, October 2001. IEEE CS Press, California, USA.
- [21] J.N. Oliveira. An introduction to point-free programming, 1999. Departamento de Informática, Universidade do Minho. 37p., chapter of book in preparation.
- [22] Alberto Pardo. Fusion of recursive programs with computational effects. *Theoretical Computer Science*, 260(Issue 1-2):165–207, June 2001.
- [23] Alberto Pardo. Combining datatypes and effects. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming*, volume 3622 of *LNCS*, pages 171–209. Springer Verlag, 2005.
- [24] G. Villavicencio. Reverse program calculation by conditioned slicing. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*, pages 368–378, Benevento, Italy, March 2003. IEEE CS Press, California, USA.