Universidad Católica de Santiago del Estero

Facultad de Matemática Aplicada

Technical Report TR-804

Handling State in Reverse Program Calculation

Gustavo Villavicencio

Campus de la UCSE
gustavov@ucse.edu.ar

Abstract

The Reverse Program Calculation is the process by which starting from imperative source code we go up to more abstract representation, usually expressed in point-free style. It is a really calculational process. In previous works, we have used pure functional expressions as support for denotational sematics, avoiding the treatment of side effects. In this paper we are treating with the monad **State** in the reverse program calculation process. This monad is particularly interesting, since it involves a implicit paramenter, the state, which becomes the reverse calculation process a bit more complicated. To this purpose, new operators, properties and laws are introduced in the context of the reverse calculation process.

1 Introduction

The Reverse Program Calculation (RPC) or reverse program specification [9, 12] supported by slicing techniques [2, 15] et al., have been treated in pure functional setting. However, the real world requieres to deal with complex situations which becomes essential the use of more sophisticated formalisms than those used in the pure functional context. Such situations, usually involve *side effects*.

That is, in previous works [9, 12], we have not attacked the difficulties that side effects involve. This was because, we have avoiding the treatement of complex situations at the begining of our research. However, in real conditions, the use of side effects is usual and so, their handle is quite important in the RPC context.

So, in real programs, not all the functions return values that only depend of the input data. To turn possible the side effect treatment in the functional context, the functional community have adopted the concept of *monad* [7, 8]. Although the idea was developed by Moggi to structure *denotational semantics*, was Wadler who introduced the concept to the functional context [13].

By means of the monad application, the critical properties of *referential transparence* and *equational reasoning* are restored, and therefore, the calculational process we are proposing is still possible.

Specifically, in this paper we are going to operate on a specific side effect, i.e. state, from the RPC perspective. This analysis is very interesting, since the monad **State** requires the use of *exponential datatypes*, and thus, of law, properties, and operators, which are not presents in other monads.

This paper is organised as follows. In section 2 we will introduce the basic concepts related to monads. Then, we will develop the monad **State** concept in section 3. The RPC strategy will be presented in section 4, where we develop the theory that will applied in an example. In the last two sections, 5 and 6, we will develop the work in front and the conclusions, respectively.

2 Monads

In the Eugenio Moggi's paper on monads [8], the main matter is distinguish functions that return a value and it only dependent of the input arguments,

from those functions that return some value as consequence of their input arguments and can also generete another results like side effects, nondeterminism, IO, etc. In certain contexts [1] et. al., monads are also reference as *triples*.

Formally, a triple $T = (T, \eta, \mu)$ in the $\mathcal{C}$ category is a (endo)functor $T : \mathcal{C} \longrightarrow \mathcal{C}$ and two *natural transformations*(polymorphic functions) $\eta : id_{\mathcal{C}} \Rightarrow T$ and $\mu : TT \Rightarrow T$ which satisfices the following properties

$$\mu \cdot \mu T = \mu \cdot T\mu \tag{1}$$
$$\mu \cdot T\eta = \mu \cdot \eta T = id_{\mathcal{C}} \tag{2}$$

(1) is called the *associative law* of a monad as (2) the *left* and *right unit* of a monad.

The context where the monads become well known was in the functional programming setting [14]. In it, monads are interpreted like a *triple kleisli*. A kleisli triple $(M, \eta, _^*)$ in a $\mathcal{C}$ category, is composed by a functor at objects level $M : Obj(\mathcal{C}) \longrightarrow Obj(\mathcal{C})$, a natural transformation $\eta : I \Rightarrow M$, and a extension operator $f^* : MA \longrightarrow MB$ where $f : A \longrightarrow MB$, such that the following properties are satisfaced:

$$\eta_A^* = id_{MA} \tag{3}$$
$$f^* \cdot \eta_A = f \tag{4}$$
$$f^* \cdot g^* = (f^* \cdot g)^* \tag{5}$$

where $f : A \longrightarrow MB$ y $g : B \longrightarrow MC$.

It is important to note, that the extension operator supplies a mechanism to compose monadic funcions, like two previously defined. That is

$$g \bullet f \stackrel{def}{=} g^* \cdot f \tag{6}$$

Based on this definition, we can determine that the laws (3) and (4), establishes that η is the right and left identity, as long as (5) establishes that the composition is associative.

Once the kleisli triple is defined, we are in condition to define the *Kleisli Category*. Let a triple kleisli $(M, \eta, _^*)$, a kleisli categeory $\mathcal{C}_M$ is defined as:

- The objects in $\mathcal{C}_M$ are the same that in $\mathcal{C}$.

- Los morfismos $\mathcal{C}_M(A,B) \equiv \mathcal{C}(A, MB)$.
- Identidad est'a dada por $\eta_A : A \longrightarrow MA$.
- La composici'on es dada por la composici'on kleisli (6).

2.1 Monads in functional programming

The previous definition of the kleisli triple, it can be expressed in the following way in the functional programming context: $(M, unit, \star)$. M is a type constructor, $unit : A \longrightarrow MA$ a polymorphic function, and $\star : MA \times (A \longrightarrow MA) \longrightarrow MA$ a polymorphic operator, usually called *bind*.

The function composition in functional programming is denoted by $m \star f$, what 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: $m \star \lambda v.f$.

As example, we take the **Exception monad**, i.e. which model exceptions in a program. Let E the type of exception values produced by a program, then $MA = A + E$ is the type that model successful computations, or fail to generating exceptions of type E.

Regarding $unit_A : A \longrightarrow A + E$, it can be modelled by the injection i_1. The extension operator applicable on a function $f : A \longrightarrow MB$, meanwhile, it can modelled as $_^* = [_, i_2]$. That is

$$\begin{array}{lcl} f^* & : & MA \longrightarrow MB \\ f^* & = & [f, i_2] \end{array} \tag{7}$$

In particular, and because in this work we are interested in the monad **State**

3 State

In this section we are going to treat how the state are handling in a functional setting. But before, some basic concepts are needed.

3.1 Exponentials

As we known, in cases where $f : C \times A \longrightarrow B$ and $g : A \longrightarrow B$, we can use *split* operator but recurriendo to a previous *interface*.

$$
\begin{array}{lcl}
f_c & : & A \longrightarrow B \\
f_c a & \stackrel{def}{=} & f(c,a)
\end{array}
\tag{8}
$$

That is, making irrelevent the first argument of the function domain. In consequense, we can now to compose the previous functions by mean of the split opetator: $\langle f_c, g\rangle$.

It is important take in mind that, for $f : C \times A \longrightarrow B$, fc denote a value of type B (that is $fc \in B$), mientras que f_c denote a function of type $A \longrightarrow B$ (that is $f_c \in B^A$), where

$$
B^A = \{g | g : A \longrightarrow B\}
\tag{9}
$$

construction which is called *exponential*.

Based on this new data type, we can define a new operator that take a function and apply it. It is know, as *application operator*.

$$
\begin{array}{lcl}
ap & : & B^A \times A \longrightarrow B \\
ap(g,a) & = & g\ a
\end{array}
\tag{10}
$$

Returning again to the functions $f : C \times A \longrightarrow B$, we can think on the operation that for each $c \in C$ it generate $f_c \in B^A$. It can be seen like a families of functions indexed by c, that could be denote as

$$
\begin{array}{lcl}
\bar{f} & : & C \longrightarrow B^A \\
(\bar{f}c)a & = & f(c,a)
\end{array}
\tag{11}
$$

From here we can say that, $\bar{f}$ is more “tolerat” than f since not requires both arguments at the same times for its evaluation.

Some laws are satisfes by exponentials

- **Cancellation**
$$
f = ap \cdot (\overline{f} \times id)
\tag{12}
$$

- **Reflexion**
$$
\overline{ap} = id_{B^A}
\tag{13}
$$

- **Fusion**
$$
\overline{g \cdot (f \times id)} = \overline{g} \cdot f
\tag{14}
$$

- **Absortion**
$$
\overline{f \cdot g} = f^A \cdot \overline{g}
\tag{15}
$$

- **Functor**

$$(f \cdot g)^A = f^A \cdot g^A \tag{16}$$

- **Functor-id**

$$id^A = id \tag{17}$$

In (15) we can note the use of a new operator, which because f is involved we can call it "composed with f". More precisely, $f : B \longrightarrow C$ and $f^A : B^A \longrightarrow C^A$, so f^A can accept $g : A \longrightarrow B$ as an input function and defined it as

$$(f^A)g \stackrel{def}{=} f \cdot g \tag{18}$$

3.2 Monad State

One of the more difficult monad to handle is the monad *State*, since it involves an additional argument: The state. 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

$$f \times id_S : A \times S \longrightarrow B \times S \tag{19}$$

By other side, as usual, the monad is represented algebraically like a triple

$$\mathbf{S} = (M, \eta, \mu) \tag{20}$$

with the following types for each components

$$\begin{aligned} MA &: S \longrightarrow A \times S \; and \; also \\ &: (A \times S)^S && (21) \\ \eta_A &: A \longrightarrow (A \times S)^S && (22) \\ \mu &: ((A \times S)^S \times S)^S \longrightarrow (A \times S)^S && (23) \end{aligned}$$

Now, returning our view on (19) we must note that

$$\begin{aligned} & A \times S \longrightarrow B \times S \\ \cong & \\ & A \longrightarrow (S \longrightarrow B \times S) \\ = & \\ & A \longrightarrow (B \times S)^S \end{aligned}$$

So, f is a monadic function, i.e. a state monadic function. Therefore, the morphism $f : A \longrightarrow (B \times S)^S$ in $\mathcal{C}$, related to the $f : A \longrightarrow B$ in $\mathcal{C}_{\mathcal{M}}$, can be constructed by means of (11)

$$\overline{f \times id_S} : A \longrightarrow (B \times S)^S \tag{24}$$

i.e. the f transpose is the state monadic function.

We can go further on (24) and apply it the extension operator from the kleisli triple, and note that

$$(\overline{f \times id_S})^* : (A \times S)^S \longrightarrow (B \times S)^S \tag{25}$$

Going further on (25) we can observe that

$$(\overline{f \times id_S})^* = (f \times id_S)^S \tag{26}$$

by means of (18).

Regarding to η definition, a first aproximation can be expressed by λ notation

$$\eta_A = \lambda s.(a, s) \tag{27}$$

But here, we are interested in a point-free version. By (22) we know the type of η. At the same time we know that $(A \times S)^S \cong A^S \times S^S$, so we can rewrite (27) as

$$unit_A : A \longrightarrow A^S \times S^S \tag{28}$$

from where we can establish that

$$\begin{aligned} \bar{\pi_1} &: A \longrightarrow A^S \\ \bar{\pi_2} &: A \longrightarrow S^S \end{aligned}$$

It is because, starting from (11) and proceeding in reverse order, we find that

$$\begin{aligned} \pi_1 &: A \times S \longrightarrow A \\ \pi_2 &: A \times S \longrightarrow S \end{aligned}$$

In consequence, we can determine that

$$unit_A = \langle \overline{\pi_1}, \overline{\pi_2} \rangle \tag{29}$$

or what would be the same $unit_S = \overline{\langle \pi_1, \pi_2 \rangle}$.

With regard to μ definition, we proceed as follows. As we can notice, we

have that, so much in the domain as in the codomain of μ there are exponentials. As Already have been indicated, the implicit operator is the function application (10). Well then, if we replace $B = A \times S$ and $A = S$ in (10), we would get a particularizaci'on of (10) with the new data type in which we are interested

$$\begin{array}{lcl} ap & : & (A \times S)^S \times S \longrightarrow A \times S \\ ap(f, s) & = & f s \end{array} \tag{30}$$

So, (30) with exception of the outer exponent is almost (23). In consequence, on the base of the ap operator, we can obtain a new one with the types required by μ, as follows

$$\begin{array}{lcl} (\overline{ap})^* & : & ((A \times S)^S \times S)^S \longrightarrow (A \times S)^S \\ (\overline{ap})^*(f, s) & = & \overline{ap_{S,A\times S} \cdot ap_{S,SA\times S}} \end{array} \tag{31}$$

So, ap^S is a functional operator that allow "compose with ap" (18). Obviously, we know by (26) that $(\overline{ap})^* = ap^S$.

It is clear also, that from (30) we can to obtain the transpose that would have the following signature

$$\overline{ap} : (A \times S)^S \longrightarrow (A \times S)^S \tag{32}$$

from where we establish that

$$id_{(A\times S)^S} = \overline{ap} \tag{33}$$

the identity functor is the transpose of the function application.

4 The reverse program calculation process

To make easy the understanding of the RPC process, usually we compare it with the *Laplace transformation*. Based on this constrast, we construct a schematic view shown in graph 1

Starting from pointwise expressions, usually written in **HASKELL**[6] or **VDM-SL**[5], we try to obtain an equivalent point-free expression. To reach this target, succesive transformation steps are performed on the pointwise expressions. The process finish when we obtain an expression that is handle by some property or law accesible from a calculus. Commonly, to become clear what is the law or property involved, a commutative diagram is constructed

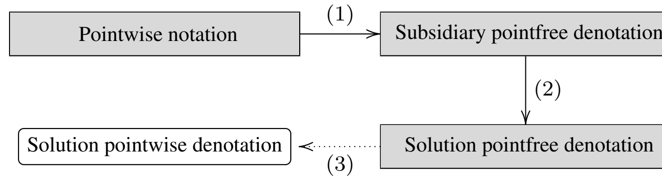

Figure 1: Laplace approach applied to RPC process

from the last pointwise expression obtained. Why, we are translating pointwise expressions in point-free expressions? Because the reasoning process is more easy on point-free expressions, and because this kind of denotation is more compact and abstract than pointwise denotation. This is the process represented by arrow **(1)** in graph 1.

Once in the point-free side, the real calculational process is performed. From the commutative diagram constructed in the previous phase, we extract a point-free expression which is summited to the applications of laws and properties accesible from a 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 graph 1.

At this point, we have calculated a very compact and abstract program representation, and therefore, a formal reverse engineering process have been outlined. To this process, we call it *Reverse Program Calculation* (RPC). However, to turn more useful the RPC process, would be desirable to re-write the calculated solution as pointwise expressions again. Furthermore, due to the calculated solution is a formal specification of the program fragment analysed, we can proceed also, with a re-implementation in a different support than the original. So, following any way, the whole process becomes a formal reengineering process. This last phase is represented by arrow **(3)** in graph 1.

Unfortunately, in the present state of our research the described process is a bit more complicated in practice. The reader will can appreciate some application difficulties during the developement of an example in section 4.4. These will be descussed in detail in section 5.

In order to be possible to develop an example, in the following sections we will introduce the theory which will be applied to it. Other laws, properties and operators have already been presented in [9, 12].

4.1 Monad state properties

Although in section 3 we have introduced the monad **State** (20), we have not proved that it is a monad really. Therefore, based on the previous definitions in section 3, we are going to prove the usual laws (3)-(5) for the monad **State**. However, before the proofs, it is necessary some other definitions. First, we must indicate that instead operate on $f : A \longrightarrow (B \times S)^S$ in the kleisli category $\mathcal{C}_M$, we are going to operate on the related morphism $f : A \longrightarrow B$ in the category $\mathcal{C}$. In this way

$$\begin{aligned} unit &= \langle\overline{\pi_1}, \overline{\pi_2}\rangle && (34) \\ _^* &= (_)^S && (35) \\ \mu &= ap^S && (36) \end{aligned}$$

where there is important to note, that the lifting operator $(_)^S$, it is applied to the previous to the transpose. Finally, and before to continue with the proof of the monadic laws, we prove that $Mf = (unit_B \cdot f)^*$ as follow

$$\begin{aligned} & Mf \\ = & \quad \{ \text{ unit definition (34)} \} \\ & (\langle\overline{\pi_1}, \overline{\pi_2}\rangle \cdot f)^* \\ = & \quad \{ \times\text{-absortion} \} \\ & \langle\overline{\pi_1} \cdot f, \overline{\pi_2} \cdot f\rangle^* \\ = & \quad \{ \text{ exponentials fusion (14)} \} \\ & \langle\overline{\pi_1 \cdot (f \times id_S)}, \overline{\pi_2 \cdot (f \times id_S)}\rangle^* \\ = & \quad \{ \text{ by (26)} \} \\ & \langle\pi_1 \cdot (f \times id_S), \pi_2 \cdot (f \times id_S)\rangle^S \\ = & \quad \{ \text{ projections promete through product} \} \\ & (\langle f \cdot \pi_1, id_S \cdot \pi_2\rangle)^S \\ = & \quad \{ \times\text{-fusion in reverse} \} \\ & ((f \times id_S) \cdot \langle\pi_1, \pi_2\rangle)^S \\ = & \quad \{ \text{ by (16), identity and (26)} \} \\ & (\overline{f \times id_S})^* && (37) \end{aligned}$$

Subsequently, we show how a function f is lifting on the base of the previous definitions, that is, the property $f^* = \mu \cdot Mf$, where $f : A \longrightarrow MA$.

$$
\begin{array}{ll}
 & \mu \cdot Mf \\
= & \qquad \{ \text{ definiciones (36) y (37)} \} \\
 & (ap^S) \cdot (f \times id_S)^S \\
= & \qquad \{ \text{ Exponentials-functor} \} \\
 & (ap \cdot (f \times id_S))^S \\
= & \qquad \{ \text{ by (26)} \} \\
 & (\overline{ap \cdot (f \times id_S)})^* \\
= & \qquad \{ \text{ by (14)} \} \\
 & (\overline{ap} \cdot f)^* \\
= & \qquad \{ \text{ by (13) and identity} \} \\
 & f^*
\end{array} \tag{38}
$$

At this point, we are in conditions to prove the monadic laws as follow. We prove (3)

$$
\begin{array}{ll}
 & unit^* \\
= & \qquad \{ \text{ definition} \} \\
 & (\overline{\langle \pi_1, \pi_2 \rangle})^* \\
= & \qquad \{ \times\text{- identity} \} \\
 & (\overline{id_{A \times S}})^* \\
= & \qquad \{ \text{ by (26)} \} \\
 & id_{(A \times S)^S}
\end{array} \tag{39}
$$

We prove (4)

$$
\begin{aligned}
& f^* \cdot unit \\
= & \quad \{ \text{ lifting and (34)} \} \\
& (f \times id_S)^S \cdot \langle \overline{\pi_1}, \overline{\pi_2} \rangle \\
= & \quad \{ \times\text{- absortion} \} \\
& \langle (f \times id_S)^S \cdot \overline{\pi_1}, (f \times id_S)^S \cdot \overline{\pi_2} \rangle \\
= & \quad \{ \text{ by (15) in reverse} \} \\
& \langle \overline{(f \times id_S) \cdot \pi_1}, \overline{(f \times id_S) \cdot \pi_2} \rangle \\
= & \quad \{ \times\text{-absortion in reverse} \} \\
& \overline{(f \times id_S) \cdot \langle \pi_1, \pi_2 \rangle} \\
= & \quad \{ \text{ identity} \} \\
& \overline{f \times id_S} \\
= & \quad \{ \text{ and by (24) we know that} \} \\
& f \; in \; \mathcal{C}_{\mathcal{M}}
\end{aligned} \tag{40}
$$

Finally we prove (5)

$$
\begin{array}{ll}
 & f^* \cdot g^* \\
= & \quad \{ \text{ lifting operator} \} \\
 & (f \times id_S)^S \cdot g^* \\
= & \quad \{ \text{ by (16)} \} \\
 & (f^S \times id_{S^S}) \cdot g^* \\
= & \quad \{ \text{ product definition} \} \\
 & \langle f^S \cdot \pi_1^S, id_{S^S} \pi_2^S \rangle \cdot g^* \\
= & \quad \{ \times\text{-fusion} \} \\
 & \langle f^S \cdot \pi_1^S \cdot g^*, id_{S^S} \cdot \pi_1^S \cdot g^* \rangle \\
= & \quad \{ \text{ lifting operator again} \} \\
 & \langle f^S \cdot \pi_1^S \cdot (g \times id_S)^S, id_{S^S} \cdot \pi_2^S \cdot (g \times id_S)^S \rangle \\
= & \quad \{ \text{ (16) in reverse} \} \\
 & \langle f^S \cdot (\pi_1 \cdot (g \times id_S))^S, id_{S^S} \cdot (\pi_2 \cdot (g \times id_S))^S \rangle \\
= & \quad \{ \text{ projections promote through product} \} \\
 & \langle f^S \cdot (g \cdot \pi_1)^S, id_{S^S} \cdot (id_S \cdot \pi_2)^S \rangle \\
= & \quad \{ \text{ (16) again} \} \\
 & \langle f^S \cdot g^S \cdot \pi_1^S, id_{S^S} \cdot id_{S^S} \cdot \pi_2^S \rangle \\
= & \quad \{ \times\text{-fusion in reverse} \} \\
 & (f^S \cdot id_{S^S}) \cdot \langle g^S \cdot \pi_1^S, id_{S^S} \cdot \pi_1^S \rangle \\
= & \quad \{ \times\text{-fusion in reverse} \} \\
 & (f^S \times id_{S^S}) \cdot (g^S \times id_{S^S}) \cdot \langle \pi_1^S, \pi_1^S \rangle \\
= & \quad \{ \text{ exponentials distribute over product and identity} \} \\
 & (f \times id_S)^S \cdot (g \times id_S)^S \cdot id_{(A \times S)^S} \\
= & \quad \{ \text{ liftign operator} \} \\
 & (f^* \cdot g^*) \cdot id_{(A \times S)^S} \\
= & \quad \{ \text{ g is in } \mathcal{C} \text{ and identity} \} \\
 & (f^* \cdot g)^*
\end{array}
$$

Once proved that (20) is a monad, we can go further and to introduce the theory required to handle recursive functions which generate also, side effects.

4.2 Distribution laws

A monad M on a category $\mathcal{C}$ with product is called *strong* if it comes equipped with a natural transformation

$$\begin{aligned} \tau_{A,B} &: A \times MB \longrightarrow M(A \times B) \\ \tau(a, m) &= do\{x \leftarrow m; return(a, x)\} \end{aligned} \tag{41}$$

(*do notation* is a syntactic sugar used in **HASKELL** to use monads easier) i.e. $\tau_{A,B}$ transforms a pair value-computation into a computation of a pair of values [8]. This natural transformation is called *strength* and satisfies the following equations [10]

$$M\pi_2 \cdot \tau_{1,A} = \pi_2 \tag{42}$$

$$\tau_{A,B\times C} \cdot (id_A \times \tau_{B,C}) \cdot \alpha_{A,B,C} = M\alpha_{A,B,C} \cdot \tau_{A\times B,C} \tag{43}$$

where π_2 is a projection and α is a *natural isomorphism* [11]. Based on the strong functor concept, we can define a *strong monad* [8].

Another important concept whose usefulness will become clear in the following sections is that of *monadic extension* of functor F in a category $\mathcal{C}$. It is a construction of type $\widehat{F} : \mathcal{C}_M \longrightarrow \mathcal{C}_M$ such that $\widehat{F}A = FA$, i.e. the objects in $\mathcal{C}$ and $\mathcal{C}_M$ are the same; and on monadic morphisms $f : A \longrightarrow MB$, it yields $\widehat{F}f : \widehat{F}A \longrightarrow M(\widehat{F}B)$, or what is the same $\widehat{F}f : FA \longrightarrow M(FB)$ in $\mathcal{C}_M$.

Closely related to the monadic extension concept is the so called *distribution law* [4]. It determines that every monadic extension $\widehat{F}$ is related one-to-one to a natural transformation $\delta^F : FM \Rightarrow MF$. This natural transformation performs the distribution of a monad over a functor. Pictorically

$$\widehat{F}f = FA \xrightarrow{Ff} FMB \xrightarrow{\delta^F_B} MFB \tag{44}$$

For instance, we can define the distribution law for the sum operator

$$\begin{aligned} \delta(A, B) &: M\ A + M\ B \longrightarrow M(A + B) \\ \delta^+_{(A,B)} &= [Mi_1, Mi_2] \end{aligned} \tag{45}$$

From here, we can derive a monadic extension to the sum functor. Given $f + g$ and composing with (45), we get

$$
\begin{aligned}
&= [Mi_1, Mi_2] \cdot (f + g) \\
&\quad + - absortion\ law \\
f \widehat{+} g &= [Mi_1 \cdot f, Mi_2 \cdot g] \qquad (46)
\end{aligned}
$$

So, $f \widehat{+} g$ denotes the monadic extension of the $+$ functor.

As example of distribution law, we can calculate the distribution law for the list base functor. It will have type $1 + A \times ML \longrightarrow M(1 + A \times L)$. Using the natural transformation defined by (41), we proceed as follows

$$
\begin{aligned}
& [Mi_1, Mi_2] \cdot (id + \tau_{(A,L)}) \\
= & \qquad \{ \text{ action of monad M in } \mathcal{C}_M \} \\
& [(unit \cdot i_1)^*, (unit \cdot i_2)^*] \cdot (id + \tau_{A,L}) \\
= & \qquad \{ \text{ either of monadic function} \} \\
& [unit \cdot i_1, unit \cdot i_2]^* \cdot (id + \tau_{A,L}) \\
= & \qquad \{ \text{ Kleisli composition definition (6)} \} \\
& [unit \cdot i_1, unit \cdot i_2] \bullet (id + \tau_{A,L}) \\
= & \qquad \{ \text{ lifting functor on injections} \} \\
& [\widehat{i_1}, \widehat{i_2}] \bullet (id + \tau_{A,L}) \\
= & \qquad \{ \text{ +-absortion} \} \\
& [\widehat{i_1}, \widehat{i_2} \bullet \tau_{A,L}] \qquad (47)
\end{aligned}
$$

This just calculated law will be used afterwards in an example.

4.3 Catas and homos

From a simple view, a *monadic fold* is a function that behaves like a fold, but with the added property of producing effects. As approximation to its definition we consider the next diagram in the Kleisli category C_M

$$
\begin{array}{ccc}
M\ A & \xleftarrow{\ h\ } & F\ A \\
\downarrow f^* & & \downarrow \widehat{F} f \\
M\ B & \xleftarrow[\ h'^*\ ]{} & M\ F\ B
\end{array}
\qquad (48)
$$

In this view, we are thinking of functions that involve a recursive process during which side effects can be produced. From (48) we can infer property

$$f \bullet h = h' \bullet \widehat{F} f \qquad (49)$$

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 $(M\ T, \widehat{in_T})$ is the initial monadic algebra, then there is a unique homomorphism to any monadic algebra $(M\ B, f)$. In a diagram

$$\begin{array}{ccc} M\ T & \xleftarrow{\widehat{in_T}} & F\ T \\ {\scriptstyle (\!|f^*|\!)}\downarrow & & \downarrow{\scriptstyle \widehat{F}(\!|f^*|\!)} \\ M\ B & \xleftarrow[f^*]{} & M\ F\ B \end{array} \qquad (50)$$

Thus the following property:

$$h = (\!|f^*|\!) \iff h \bullet \widehat{in_T} = f \bullet \widehat{F} h \qquad (51)$$

So, the *monadic fold* operator [4], $(\!|f|\!)_F^M : T \longrightarrow M\ B$ is then defined as the least homomorphism between $\widehat{in}$ and f [10]. However, we know that $h \bullet \widehat{in_F} = h \cdot in_F$ and $\widehat{F} h = \delta_B^F \cdot F\ h$, and therefore we can rewrite (51) as

$$h \cdot in_F = (f \bullet \delta_B^F) \cdot F\ h \qquad (52)$$

which pictorically would be

$$\begin{array}{ccc} T & \xleftarrow{in} & F\ T \\ & & \downarrow{\scriptstyle F(\!|f|\!)} \\ {\scriptstyle (\!|f|\!)}\downarrow & & F\ M\ B \\ & & \downarrow{\scriptstyle \delta_B^F} \\ M\ B & \xleftarrow[f^*]{} & M\ F\ B \end{array} \qquad (53)$$

In this way, every homomorphism $f : T \longrightarrow M\ A$ between the monadic algebras $\widehat{in_T}$ 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 [10].

4.4 An example

In figure 2, we shown a **HASKELL**program that sum a sequence of numbers, and return a pair (result,length of the sequence). Function *tick* increment the state each time a computation is performed. *get* and *put* are methods of the *MonadState* class. Therefore, we are using the monad **State** to know the length of the sequence of numbers.

```
tick :: State Int Int
tick = do c <- get       -- takes the state
          put (c+1)      -- replace the state
          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 monad State

From figure 2, we would want to construct a commutative diagram as (53) as shown by (54).

$$
\begin{array}{ccc}
L & \xleftarrow{\;in\;} & 1 + Int \times L \\
 & & \big\downarrow{\scriptstyle id+id\times(\!|sms|\!)} \\
\big\downarrow{\scriptstyle (\!|sws|\!)} & & 1 + Int \times (Int \times S)^S \\
 & & \big\downarrow{\scriptstyle \delta^L_{Int}} \\
(Int \times S)^S & \xleftarrow[\overline{(h \times id_S)}^*]{} & ((1 + Int \times Int) \times S)^S
\end{array}
\qquad (54)
$$

From here we can extract the next property

$$
\begin{aligned}
& sms \cdot in \\
= & \quad \{ \text{ commutative diagram} \} \\
& \overline{h \times id_S} \bullet \delta^L_{Int} \cdot (id + id \times sms) \\
= & \quad \{ \text{ distribution law definition (47)} \} \\
& \overline{h \times id_S} \bullet [\widehat{i_1}, \widehat{i_2} \bullet \tau_{Int,L}] \cdot (id + id \times sms) \\
= & \quad \{ \text{ kleisli composition definition (6)} \} \\
& (\overline{h \times id_S})^* \cdot [\widehat{i_1}, \widehat{i_2} \bullet \tau_{Int,L}] \cdot (id + id \times sms) \\
= & \quad \{ \text{ by (26)} \} \\
& (h \times id_S)^S \cdot [\widehat{i_1}, \widehat{i_2} \bullet \tau_{Int,L}] \cdot (id + id \times sms) \\
= & \quad \{ \text{ +-fusion} \} \\
& [(h \times id_S)^S \cdot \widehat{i_1}, (h \times id_S)^S \cdot \widehat{i_2} \bullet \tau_{Int,L}] \cdot (id + id \times sms) \\
= & \quad \{ \ \widehat{i_j} = \overline{i_j} \} \\
& [(h \times id_S)^S \cdot \overline{i_1}, (h \times id_S)^S \cdot \overline{i_2} \bullet \tau_{Int,L}] \cdot (id + id \times sms) \\
= & \quad \{ \text{ exponential absorption in reverse (15)} \} \\
& [\overline{(h \times id_S) \cdot i_1}, (\overline{(h \times id_S) \cdot i_2}) \bullet \tau_{Int,L}] \cdot (id + id \times sms) \\
= & \quad \{ \text{ h structure} \} \\
& [\overline{([\underline{0}, +] \times id_S) \cdot i_1}, (\overline{([\underline{0}, +] \times id_S) \cdot i_2}) \bullet \tau_{Int,L}] \cdot (id + id \times sms) \\
= & \quad \{ \text{ using left distribution} \} \\
& [\overline{([\underline{0} \times id_S, + \times id_S] \cdot distl) \cdot i_1}, (\overline{([\underline{0} \times id_S, + \times id_S] \cdot distl) \cdot i_2}) \bullet \tau_{Int,L}] \cdot (id + id \times sms) \\
= & \quad \{ \text{ distl application} \} \\
& [\overline{\underline{0} \times id_S}, (\overline{+ \times id_S}) \bullet \tau_{Int,L}] \cdot (id + id \times sms) \\
= & \quad \{ \text{ kleisli composition again (6)} \} \\
& [\overline{\underline{0} \times id_S}, (\overline{+ \times id_S})^* \cdot \tau_{Int,L}] \cdot (id + id \times sms) \\
= & \quad \{ \text{ +-absorption and identity} \} \\
& [\overline{\underline{0} \times id_S}, (\overline{+ \times id_S})^* \cdot \tau_{Int,L} \cdot (id \times sms)]
\end{aligned}
$$

by either structural equality

$$\begin{array}{lcl} sms \cdot Nil & = & \overline{\underline{0} \times id_S} \\ sms \cdot cons & = & (\overline{+ \times id_S})^* \cdot \tau_{Int,L} \cdot (id \times sms) \end{array} \tag{55}$$

So, we have calculated a solution, i.e. a formal specification for the fragment which is being analysed.

5 Future works

As have been presented, our strategy have some limitation. Regarding the refactoring process applied on pointwise expressions (arrow **(1)**), we have not defined common transformation rules in order to construct generalized transformation schemes. We expect that these transformation schemes can be implemented in a tool to become such process an automatic or semi-automatic one.

During the calculational process, an important number of laws and properties can be used in each step. Therefore, the process can become difficult to be handle by the reverse engineer. So, an automatic assitant is required to suggest to the professional which laws or properties can be applied at each moment.

With relation to theoretical aspects, we are faced to resolve more complex cases as those where more than one monad is involved, i.e. situations involving monads transformers. Preciselly, under real conditions the application of monad transformers is very usual. Although we have not definitive results, some experiments performed let us assume that the RPC complexity increase in an importante rate.

6 Conclusions

In this paper we have shown how to handle monad **State** in the context of the reverse program calculation process. To this end, we have introduced new operators, properties, and laws, which have never been treated before in this context.

In the future, some problems remain unresolved. Regarding the refactoring process, we haven't identified transformation rules to be applied through this process. In the same direction, generic transformation schemes remain also not identified. All of theses matters, are focused on the automatic support

definition to become practise the RPC process.

Regarding with more theoretical aspects, we must to begun the treatment of monad transformers. In fact, the presence of a combination of monads is very common in real applications, thus; this topic becomes important. According with the knowlodge we have at the moment, one way to follows to handle this topic would be *adjunctions* [3].

References

[Barr and Wells2002] Barr, M. and Wells, C. (2002) , *Toposes, Triples and Theories*, Revised version - Version 1.1

[Canfora et al.1998] Canfora, G., Cimitile, A., and Lucia, A. D. (1998) , *Information and Software Technology (special issue on program slicing)* **40(11/12)**, 595

[Fokkinga and Meertens1993] Fokkinga, M. and Meertens, L. (1993) , ...

[Fokkinga1994] Fokkinga, M. M. (1994) , *Monadic Maps and Folds for Arbitrary Datatypes*, Technical Report Memoranda Inf 94-28, Enschede, Netherlands: University of Twente

[IFAD1999] IFAD (1999) , *VDM Tools*, Technical report, Forskerparken 10, DK-5230 Odensen M, Denmark: IFAD, `http://www.ifad.dk`

[Jones2003] (Jones, S. P. ed.) (2003) , *Haskell 98 Language and Libraries*, Cambridge, UK: Cambridge University Press

[Moggi1989] Moggi, E. (1989) , In *IEEE Symposium on Logic in Computer Science*, pp. 14–23

[Moggi1991] Moggi, E. (1991) , *Informations and Computations* **93(1)**, 55

[Oliveira and Villavicencio2001] Oliveira, J. N. and Villavicencio, G. (2001) , In *Proceedings of the 8th Working Conference on Reverse Engineering*, pp. 35–45, IEEE CS Press, California, USA

[Pardo2001] Pardo, A. (2001) , *Theoretical Computer Science* **260(Issue 1-2)**, 165

[Simpson et al.2003] Simpson, A., Bucalo, A., and Führmann, C. (2003) , *Theoretical Computer Science* **294**, 31

[Villavicencio2003] Villavicencio, G. (2003) , In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*, pp. 368–378, IEEE CS Press, California, USA

[Wadler1992] Wadler, P. (1992) , In *19'th Symposium on Principles of Programming Languages*, Albuquerque: ACM Press

[Wadler1995] Wadler, P. (1995) , In *Advanced Functional Programming*, No. 925 in LNCS, Springer Verlag

[Weiser1981] Weiser, M. (1981) , In *Fifth International Conference on Software Engineering, San Diego*