Universidad Católica de Santiago del Estero

Facultad de Matemática Aplicada

Technical Report TR-504

Monadic Refactoring by Reverse Program Calculation

Gustavo Villavicencio

Campus de la UCSE
Santiago del Estero, Argentina
gustavov@ucse.edu.ar

Abstract

In the context of a program analysis approach based on reverse program calculation (RPC) we start from HASKELL or VDM-SL descriptions of the denotational semantics of imperative programs. The need to treat side effects requires more sophistication than that of pure functional denotations. We follow the common practice in the HASKELL community of using the algebraic concept of a monad in treating side effects, thus preserving referential transparency and equational reasoning, which are central to our formal RPC approach.

In this paper we show how to proceed in the presence of side effects affecting recursive computations. We show that other kinds of transformation are required for such functional expressions to be formally treatable. In this context, new algebraic laws are applied to obtain compact point-free expressions.

1 Introduction

In previous work [Oliveira and Villavicencio2001] we have used **HASKELL** as support for the *denotational semantics* of programs written in imperative style (i.e. **C** language). In **HASKELL**, as in ordinary mathematics, the output of the functions are solely determined by their input values. Unfortunately, this imposes severe limitations in modelling program with side effects.

So, for example, the concept of *change of state* used to explain how a program is executed is not functional. A program in state t_i mapping a variable x to the value v_x, can also map x to another value v_y in the next state t_j. Therefore, given the same input x, the result returned by some function 'value of' is not the same.

The **IO** operations are other kind of computations whose outputs are not determined solely by their inputs. A function that gets a char from a file will return a different char the next time it is called since the state of the file has changed. A solution would be to generate a new (updated) state together with the output, thus enabling the function to operate subsequently over the new state (*state-passing style*).

Another important class of operations include computations which, instead of passing an updated state, will require the passing of an $exception$, for example. The form in which they are written is called *exception-passing style*.

To the previous situations we can add others where the same conditions are present: *nondeterminism*, *error handling*, *concurrency*,etc. All of these are known as *computational effects* [Benton et al.2000]. The functions that exhibit behavior with computational effects affect the referential transparency and the equational reasoning which are critical properties of pure functional languages.

To restore these properties, a natural procedure would seem to be limiting the scope of computational effects. **Monads** [Moggi1989, Moggi1991] can be used for this purpose. Moggi [Filinski1996] was among the first to observe that effects can be modelled as particular instances of a generic schema, parameterized by a monad. That is, the theory of computational effects can be derived in an abstract way without reference to any particular effect.

This perspective provides an abstract and uniform way of modelling features in programming languages, thanks to monadic structures that hide the details on how computational effects are internally represented and composed.

However, it was Wadler [Wadler1990, Wadler1995] who observed that monads can also be applied to structure functional programs and not only the underlying semantics. Wadler's perspective was quickly adopted by the functional community turning monads into a core device in modern computer

programming.

In the case of **HASKELL**, the use of monads is further encouraged by the use of the *do-notation*. Unfortunately, the ease in the application of monadic concepts, combined with the unawareness programmers have on the formal underlying theory, affect correct program writing and re-structuring. There is little doubt that this disturbs program understanding.

Thus, in this paper we are particularly interested in dealing with refactoring monadic structures, with the aim of putting some order into the underlying denotational semantics that allows to carry on the process of *inverse calculus* [Oliveira and Villavicencio2001, Villavicencio2003]. The idea is to apply monadic refactoring techniques until we obtain expressions which are treatable with some law available from a calculus.

We recall from [Oliveira and Villavicencio2001] that, after translating the denotational semantics into **HASKELL**, a series of transformations are performed to pave the way for the application of algebraic laws. After that, the calculation process continues on the "algebraic side", until sufficiently abstract formulae are calculated. So, precise and compact program specifications are obtained by mathematical means. Usually, the specifications calculated are expressed in *point-free style* [Cunha and Pinto2004, Oliveira1999].

Although in [Villavicencio2003] we have used **VDM-SL**as semantic support, we could have used **HASKELL** too, and vice-versa. Regardless of which support is used, our emphasis is put on semantic transformation. Its complexity arises not only from the intrinsic algorithmic context but also from other factors such as side effects.

Specifically, and because it is one of the most interesting situations, we focus on handling computational effects in the presence of recursion. In [Erkök2002], a new way to handle recursion in the presence of computational effects is proposed, based on the concept of the so called *value recursion*. In this approach, the main idea is that the fixpoint be calculated only over the values, without repeating or losing effects. The introduced value recursion operator $mfix$ would allow the variables to be recursively linked whenever the corresponding monad comes equipped with the appropriate fixpoint operator [Erkök and Launchbury2002].

Therefore, for a term $fix\ x.e$ where e is an expression whose evaluation has side effects and fix is some fixed point operator, there are two notions of monadic recursion [Moggi and Sabry2004]:

1. The usual unfolding semantics, where the term is equivalent to $e\{x = fix\ x.e\}$, and the side-effects are duplicated by the unfolding.

2. Value recursion, where the term is equivalent to $v\{x = fix\ x.v\}$ where

v is a value obtained after e is evaluated, and the side-effects are performed only the first time e is evaluated.

In this paper, we will use the usual unfolding semantics approach to model monadic recursion, not only because it is better known, but also because the value recursion still has open problems [Moggi and Sabry2004], specially with monadic *continuation passing style* (CPS).

In summary, the main target of this paper is to explore the RPC process in the presence of monadic effects in recursive structures: what kind of transformations are required?, is there a need for new algebraic laws in the reasoning?. In order to shorten the path to follow, the slicing techniques used in [Oliveira and Villavicencio2001, Villavicencio2003] are set aside.

In section 2 we will present the main concepts related to our area of interest, and an easy example will be developed to emphasize the target we are pursuing. In section 3, we go deeper into the previous concepts in order to handle more interesting recursive programs. Again, an example is provided.

Finally, sections 4 and 5 present work ahead and the conclusions respectively.

2 Background

Many operators like *split*($\langle , \rangle$) and *either* ($[,]$) used in this paper, have already been developed in [Oliveira and Villavicencio2001, Villavicencio2003] and in more detail in [Oliveira1999]. The properties related to these operators, can also be found in the previous references. Therefore, in this paper we will only introduce the additional algebraic theory demanded by the new context application of our approach; i.e. the presence of side effects.

2.1 Monads

A monad over a category $\mathcal{C}$ is a triple (M, η, μ) where $M : \mathcal{C} \longrightarrow \mathcal{C}$ is an endofunctor and two natural transformations: $\eta : id_{\mathcal{C}} \longrightarrow M$ and $\mu : M^2 \longrightarrow M$ which obey two laws:

$$\mu \cdot (\eta \cdot M) \equiv 1_{\mathcal{F}} \equiv \mu \cdot (M \cdot \eta) \quad (1)$$

$$\mu \cdot (\mu \cdot M) \equiv \mu \cdot (M \cdot \mu) \quad (2)$$

the former is called the *left* and *right unit of a monad* where $1_{\mathcal{F}}$ [1] is the identity

[1] $\mathcal{F}$ is a category where the objects are functors and the morphisms are natural transformations.

in $\mathcal{F}$, and the latter the associative law of a monad.

In [Moggi1989] the main idea is to distinguish functions whose returned values are solely determined by their input values from those that can produce more results than the result explicitly returned. These last are called *computational effects*.

Some notions of computational effects in the category of sets are [Moggi1991, Benton et al.2000]:

- **Partiality**: $M\ A = A_{\perp}$, where $\perp$ is the diverging computation
- **Exception** $M\ A = A + E$, where E is the set of exceptions
- **Continuations** $M\ A = R^{R^A}$, where R is the set of results

Since the concept of monad can be defined in various ways, we adopt a suitable one for our purposes, the so-called *Kleisli triple*. A *Kleisli triple* $(M, unit, \_^*)$ over a category $\mathcal{C}$ is given by the endofunctor M restricted only to objects, a natural transformation $unit : id_{\mathcal{C}} \to M$, and an extension operator $\_^*$ which takes a morphism $f : A \longrightarrow M\ B$ and "lifts" its domain to $M\ A$, i.e. $f^* : M\ A \longrightarrow M\ B$, such that the next properties hold

$$unit_A^* = id_{M\ A} \tag{3}$$

$$f^* \cdot unit_A = f \tag{4}$$

$$f^* \cdot g^* = (f^* \cdot g)^* \tag{5}$$

In this context, objects of type $M\ A$ model computations delivering values of type A. Thus, $unit_A$ is an operation that turns a value into the computations which return that value and does nothing else [Wadler1995]. On the other hand, the extension operator provides a mechanism to compose monadic functions. The Kleisli composition of two monadic functions $f : A \longrightarrow M\ B$ and $g : B \longrightarrow M\ C$ is defined as

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

Under the light of this definition, the Kleisli triple laws are interpreted as follows: the first two laws mean that $unit$ is a right and left identity with respect to Kleisli composition, whereas the last one expresses that composition is associative. Therefore, monadic morphism forms a category [Pardo2001].

So, we can define the *Kleisli category* $\mathcal{C}_M$ for each Kleisli triple $(M, unit, \_^*)$ as follows: the objects in $\mathcal{C}_M$ are the same as in $\mathcal{C}$, morphisms from A to B in $\mathcal{C}_M$ are related to arrows $A \longrightarrow M\ B$ in $\mathcal{C}$, identity is defined by

$unit_A : A \longrightarrow M\ A$, and finally, the composition by the Kleisli composition.

In order to translate objects and morphisms from $\mathcal{C}$ to $\mathcal{C}_M$ we can define a *lifting functor* $(\widehat{-}) : \mathcal{C} \longrightarrow \mathcal{C}_M$. For objects, this functor is the identity functor and for morphisms $f : A \longrightarrow B$, defined as $\widehat{f} = init_B \cdot f$. Moreover, we can define the converse of $(\widehat{-})$, $\mathcal{R} : \mathcal{C}_M \longrightarrow \mathcal{C}$. On objects, it is defined as $\mathcal{R}A = MA$, and on morphisms as $\mathcal{R}f = f^*$ where $f : A \longrightarrow MB$.

Both previous definitions are equivalent, and we can construct each one from the other. So, from the Kleisli triple components the action M on $f : A \longrightarrow B$ is defined by $Mf = (unit_B \cdot f)^*$, and $\mu_A = id^*_{MA}$. Besides, we can construct a Kleisli triple from a monad $(M, unit, \mu)$, restricting M to objects, and defining the extension of each $f : A \longrightarrow MB$ as $f^* = \mu_B \cdot Mf$.

2.2 An example

Supposing that we interested in dividing two numbers, and we want to avoid the operation of printing an error message, when the divisor is zero. A monadic version of a program that does this is shown in figure 1 where $Me\ A$ is isomorphic to $A + String$, which is an instance of $M\ A = A + E$, the already mentioned monad $Exception$.

```
type Mess =  String
data Me a  =  Res a | Exc Mess

divide      ::  Int -> Int -> Exc Int
divide x y  =
    if y==0
       then Exc "division by zero"
       else Res (x `div` y)

eval  ::  Show a => Me a -> [Char]
eval (Exc c) =  c
eval (Res a)    =  show a
```

Figure 1: Division function in **HASKELL**

Therefore, we can introduce the next definitions for this particular monad

$$
\begin{aligned}
unit &= i_1 && (7)\\
\mu &= [id, i_2] && (8)\\
f^* &= \mu \cdot (M\ f)\\
&= [id, i_2] \cdot (f + id)\\
&= [f, i_2] && (9)\\
x\ >>=\ f &= f^* x && (10)
\end{aligned}
$$

and proceed to prove (3), (4) and (5) as follows:

- Right unit: Proving (3)

$$
\begin{aligned}
& unit^*\\
= & \quad \{ \text{definition} \}\\
& i_1^*\\
= & \quad \{ \text{definition} \}\\
& [i_1, i_2]\\
= & \quad \{ +\text{-reflexion} \}\\
& id && (11)
\end{aligned}
$$

- Left unit: Proving (4)

$$
\begin{aligned}
& f^* \cdot unit\\
= & \quad \{ \text{definitions} \}\\
& [f, i_2] \cdot i_1\\
= & \quad \{ +\text{-cancellation} \}\\
& f && (12)
\end{aligned}
$$

- Associativity: Proving (5)

$$
\begin{array}{ll}
 & f^* \cdot g^* \\
= & \quad \{\ g^* \text{ definition}\} \\
 & f^* \cdot [g, i_2] \\
= & \quad \{\ +\text{-fusion}\} \\
 & [f^* \cdot g, f^* \cdot i_2] \\
= & \quad \{\ f^* \text{ definition}\} \\
 & [f^* \cdot g, [f, i_2] \cdot i_2] \\
= & \quad \{\ +\text{-cancellation}\} \\
 & [f^* \cdot g, i_2] \\
= & \quad \{\ \_^* \text{ definition}\} \\
 & (f^* \cdot g)^*
\end{array} \tag{13}
$$

Returning to our example, and applying pattern matching on the $divide$ function we can rewrite it as

$$
\begin{array}{lcl}
divide & :: & Int \to Int \to Me\ Int \\
divide\ x\ 0 & = & Exc\ "no\ divide" \\
divide\ x\ y & = & Res\ (x\ `div`\ y)
\end{array} \tag{14}
$$

Note that this function can also be written as an *uncurried function*: $A \times A \longrightarrow Exc\ A$. In point-free notation this function can be modelled as

$$
\begin{array}{l}
\quad udivide \\
= \\
\quad [i_1 \cdot (uncurried(/)) \cdot \langle \pi_1, \pi_2 \rangle, i_2 \cdot const\text{“}no\ divide''] \cdot ((= 0) \cdot \pi_2)?
\end{array} \tag{15}
$$

Because $\langle \pi_1, \pi_2 \rangle = id$, and applying $+ - absorption$ in reverse order, we obtain

$$
udivide = [i_1, i_2] \cdot (uncurried(/) + const\text{“}no\ divide'') \cdot ((= 0) \cdot \pi_2)? \tag{16}
$$

At this point, we can use *McCarthy's conditional* to rewrite the equation

$$
udivide = (= 0) \cdot \pi_2 \longrightarrow [i_1, i_2] \cdot ((uncurried(/)) + const\text{“}no\ divide'') \tag{17}
$$

Suppose now that we have a *lifting operator* $\_^*$ such that for each $f : A \longrightarrow M\ B$ (where M is a datatype like Me) yields $f^* : M\ A \longrightarrow M\ B$. By uncurrying the $divide$ function and by means of the lifting operator we have

$$udivide^* : A \times A + E \longrightarrow M\ A$$

Therefore, we could write the $udivide^*$ function in *point-free style* as follows

$$udivide^* = [((= 0)\cdot\pi_2 \longrightarrow [i_1, i_2]\cdot((uncurried(/))+const\text{“}no\ divide'')), i_2] \tag{18}$$

In order to see how (4) is fulfilled, we write

$$udivide^* \cdot i_1 = udivide \tag{19}$$

(recall (12)) where $udivide$ is defined in (17).

With regard to (3), it is easy to see that $i_1 : a \longrightarrow Me\ a$, therefore $i_1^* : Me\ a \longrightarrow Me\ a$, and in consequence $i_1^* = id_{Me}$ (recall (11)).

Suppose now that we have a function which computes the nth even number, as is shown in figure 2. Thus, we can ***compose*** $udivide$ and $dupE$ as follows

```
dupE   ::  Int -> Me Int
dupE n  =  return (2*n)
```

Figure 2: Even number function

$$\begin{array}{lcl} multdiv & :: & Int \times Int \to Me\ Int \\ multdiv(x, y) & = & udivide(x, y) >>= \backslash r- > dupEr \end{array} \tag{20}$$

where two important operators arise

$$\begin{array}{rcl} return & :: & a \to Me\ a \\ >>= & :: & Me\ a \to (a \to Me\ b) \to Me\ b \end{array}$$

While the first takes a value and generates a **computation**, the second **composes** computations. By means of (6), we could rewrite (20) as

$$multdiv = dupE \bullet udivide = dupE^* \cdot udivide$$

which is the standard Kleisli composition definition (6). So, regarding (5), it is easy to observe that the next equation is valid

$$dupE^* \cdot udivide^* = (dupE^* \cdot udivide)^* \tag{21}$$

(recall (13)).

Therefore, the triple $(Me, i_1, \_^*)$ is a **Kleisli triple**, and so a **monad**, where Me is an **endofunctor** and i_1 and $\_^*$ are **natural transformations**. In order to make this triple work like a monad, the definition in figure 3 is required in **HASKELL** where the endofunctor Me is defined as an instance of the *Monad Class*, as long as the monadic operators $return$ and $>>=$ are adapted to Me.

```
instance Monad Me where
   exc >>= f  =  case exc of
       Res a -> f a
       Exc e -> Exc e
   return a = Res a
```

Figure 3: Monad class instantiation

Altogether, we have identified all the formal (monadic) components to have a formal description of the program. All this information can be put together in a single triple

$$(Me, i_1, [udivide, i_2]) \tag{22}$$

which is the Kleisli triple that handles exceptions in our program example.

3 Monadic catas

Our example in section 2.2 is not recursive. In order to handle more interesting recursive programs, more advanced monadic concepts are needed.

3.1 Further concepts

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

$$\begin{aligned} \tau_{A,B} &: A \times MB \longrightarrow M(A \times B) \\ \tau(a, m) &= do\{x \leftarrow m; return(a, x)\} \end{aligned} \tag{23}$$

(*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 [Moggi1991]. This natural transformation is called *strength* and satisfies the following equations [Pardo2001]

$$\begin{aligned} M\pi_2 \cdot \tau_{1,A} &= \pi_2 && (24) \\ \tau_{A,B\times C} \cdot (id_A \times \tau_{B,C}) \cdot \alpha_{A,B,C} &= M\alpha_{A,B,C} \cdot \tau_{A\times B,C} && (25) \end{aligned}$$

where π_2 is a projection and α is a *natural isomorphism* [Simpson et al.2003]. Based on the strong functor concept, we can define a *strong monad* [Moggi1991].

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* [Fokkinga1994]. 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

$$\widehat{F}f = FA \xrightarrow{Ff} FMB \xrightarrow{\delta^F_B} MFB \qquad (26)$$

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

$$\begin{aligned} \delta(A,B) &: M\,A + M\,B \longrightarrow M(A+B) \\ \delta^+_{(A,B)} &= [Mi_1, Mi_2] \end{aligned} \qquad (27)$$

From here, we can derive a monadic extension to the sum functor. Given $f+g$ and composing with (27), we get

$$\begin{aligned} &= [Mi_1, Mi_2] \cdot (f+g) \\ &\quad + - absortion\ law \\ f\widehat{+}g &= [Mi_1 \cdot f, Mi_2 \cdot g] \end{aligned} \qquad (28)$$

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

3.2 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

$$\begin{array}{ccc} M\ A & \xleftarrow{\quad h \quad} & F\ A \\ {\scriptstyle f^*}\downarrow & & \downarrow{\scriptstyle \widehat{F} f} \\ M\ B & \xleftarrow[\quad h'^* \quad]{} & M\ F\ B \end{array} \tag{29}$$

In this view, we are thinking of functions that involve a recursive process during which side effects can be produced. From (29) we can infer property

$$f \bullet h = h' \bullet \widehat{F} f \tag{30}$$

where $h : FA \to MA$ and $h' : FB \to MB$ are monadic algebras and $f : A \to MB$ 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

$$\begin{array}{ccc} M\ T & \xleftarrow{\quad \widehat{in_T} \quad} & F\ T \\ {\scriptstyle (\!|f^*|\!)}\downarrow & & \downarrow{\scriptstyle \widehat{F}(\!|f^*|\!)} \\ M\ B & \xleftarrow[\quad f^* \quad]{} & M\ F\ B \end{array} \tag{31}$$

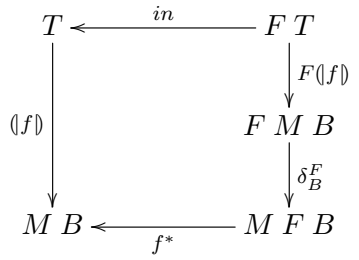
Thus the following property:

$$h = (\!|f^*|\!) \iff h \bullet \widehat{in_T} = f \bullet \widehat{F} h \tag{32}$$

So, the *monadic fold* operator [Fokkinga1994], $(\!|f|\!)_F^M : T \longrightarrow M\ B$ is then defined as the least homomorphism between $\widehat{in}$ and f [Pardo2001]. 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 (32) as

$$h \cdot in_F = (f \bullet \delta_B^F) \cdot F\ h \tag{33}$$

which pictorically would be



(34)

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 [Pardo2001].

3.3 An example

The **HASKELL** code in figure 4 takes a list and subtracts from an element the previous subtraction calculated from the rest of the list. The process is right to left. The data type Me has already been introduced in section 2.2, So, $Me\ A$ models computations that either succeed and return a value of type A or fail.

Clearly, the most interesting function is the $subtList$ function. As we know, $foldr$ is a higher order function which encapsulates structural recursion on lists, as defined in figure 5.

Let us unfold $subtList$ so that it becomes more readable:

$$\begin{array}{lcl} subtList\ [] & = & Res\ 0 \\ subtList\ (x : xs) & = & subtList\ xs\ `mcomp`\ \backslash r- > \\ & & x\ `subtM`\ r \end{array} \tag{35}$$

This version requires the $mcomp$ operator defined in figure 6. It is a *binding operator* defined for this specific monad.

More examples of this kind of transformation can be found in [Meijer and Jeuring1995]. At this point, it is useful to draw an instance of commutative diagram (34) for $subtList$:

```
type Mess = String
data Me a = Res a | Exc Mess

subtM :: Int -> Int -> Me Int
n1 `subtM` n2 =
 if n2>n1
    then Exc "Negative value calculated"
    else Res (n1-n2)

subtList l =
 foldr
   (\e1 lr ->
      case lr of
        Exc e -> Exc e
        Res a -> case (el `subtM` a) of
                   Exc e -> Exc e
                   Res a -> Res a)
   (Res 0)
   l

evalu (Exc e) = e
evalu (Res a) = show a
```

Figure 4: List subtraction example

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

Figure 5: foldr function

```
m `mcomp` f =
  case m of
    Exc e -> Exc e
    Res a -> f a
```

Figure 6: mcomp operator

$$\begin{array}{ccc} L & \xleftarrow{in} & 1 + Int \times L \\ \Big\downarrow{\scriptstyle subtList} & & \Big\downarrow{\scriptstyle id+id\times subtList} \\ M\ Int & \xleftarrow[[Res,subtM]^*]{} & 1 + Int \times M\ Int \\ & & \Big\downarrow{\scriptstyle \delta^{L_A}_{Int}} \\ & & M(1 + Int \times Int) \end{array} \tag{36}$$

However, we still haven't defined the distribution law for the list base functor, which will be of type $1 + A \times ML \longrightarrow M(1 + A \times L)$. Via (23), we proceed as follows

$$
\begin{aligned}
& [Mi_1, Mi_2] \cdot (id + \tau_{(A,L)}) \\
= & \quad \{ \text{ action of monad M in } \mathcal{C}_M \} \\
& [(unit \cdot i_1)^*, (unit \cdot i_2)^*] \cdot (id + \tau_{A,L}) \\
= & \quad \{ \text{ either of monadic function} \} \\
& [unit \cdot i_1, unit \cdot i_2]^* \cdot (id + \tau_{A,L}) \\
= & \quad \{ \text{ Kleisli composition definition (6)} \} \\
& [unit \cdot i_1, unit \cdot i_2] \bullet (id + \tau_{A,L}) \\
= & \quad \{ \text{ lifting functor on injections} \} \\
& [\widehat{i_1}, \widehat{i_2}] \bullet (id + \tau_{A,L}) \\
= & \quad \{ \text{ +-absortion} \} \\
& [\widehat{i_1}, \widehat{i_2} \bullet \tau_{A,L}] \qquad (37)
\end{aligned}
$$

Now, we are in condition to resume the exercise and infer the next property from (36)

$$
\begin{array}{ll}
 & subtList \cdot in \\
= & \quad \{ \text{ from (36)} \} \\
 & [Res, subtM] \bullet (\delta_{Int}^{L_A} \cdot (id + id \times subtList)) \\
= & \quad \{ \text{ list distribution law definition (37)} \} \\
 & [Res, subtM] \bullet ([\widehat{i_1}, \widehat{i_2} \bullet \tau_{Int,L}] \cdot (id + id \times subtList)) \\
= & \quad \{ \text{ monadic extension of +-functor (28) and associativity} \} \\
 & ([Res, subtM] \bullet (id \widehat{+} \tau_{Int,L})) \cdot (id + id \times subtList) \\
= & \quad \{ \text{ Kleisli definition (6)} \} \\
 & ([Res, subtM]^* \cdot (id \widehat{+} \tau_{Int,L})) \cdot (id + id \times subtList) \\
= & \quad \{ \text{ monadic +-absortion} \} \\
 & [Res, subtM \cdot \tau_{Int,L}] \cdot (id + id \times subtList) \\
= & \quad \{ \text{ +-absortion} \} \\
 & [Res, subtM \cdot \tau_{Int,L} \cdot (id \times subtList)]
\end{array}
$$

Structural equality enables us to rewrite this expression as

$$
\begin{array}{lcl}
subtList \cdot Nil & = & Res \\
subtList \cdot cons & = & subtM \cdot \tau_{Int,l} \cdot (id \times subtList)
\end{array}
\tag{38}
$$

which is the point-free version of the *subtList* function. Therefore, equation (38) is the formal especification (in point-free style) calculated to the *subtList* function from figure 4. Of course, the pointwise version [de Moor and Gibbons2000] can be rebuilt from this via (23), expressed in *lambda notation* [Barendregt1984, Barendregt1997] as follows

$$
\begin{array}{lcl}
subtList\ Nil & = & Res\ 0 \\
subtList(cons(a, l)) & = & subtList(l) * \lambda x.subtM(a, x)
\end{array}
$$

In **HASKELL** this amounts to the single line program shown in figure 7 where $mfold$ function is defined as shown in figure 8.

Therefore, figure 7 is the refactoring result produced as consequence of the previous reverse calculation process performed. So, we have executed a refactoring process supported by reverse calculation.

It is important to emphasize the results from equation (38) and figure 7,

```
subtList l = mfold subtM (Res 0) l
```

Figure 7: subtList function applying mfold

```
mfold :: (Monad a) => (b -> c -> a c) ->
                      a c -> [b] -> a c
mfold f k []     = k
mfold f k (x:xs) = do {b <- mfold f k xs;
                       f x b}
```

Figure 8: mfold function

since they are the main results (for the current example) of the process which is described in this paper.

4 Future and related work

The reader may have noticed that one of the most critical phases of the process illustrated above is the transformation step. As we have already pointed out, the main goal of the transformation process is to find a "bridge" heading to the point-free side. However, up to now, we do not have a precise view on how it must be performed. In other words, we do not have a standard set of (monadic) transformation rules applicable to pointwise expressions. Clearly, the purpose is to find generalized transformation schemes to make this process a mechanic one.

In [Erwig and Ren2004] a monadification process is presented to make monadic, functions that were not so previously. To attain this target, three kinds of operations are defined: *navigating*, *binding* and *wrapping*. They involve a class of monadic refactoring like the one we are interested in. However, it is insufficient for our purposes since it simply introduces effects in pure functions, but not necessarily helps to discover which formal operators, properties or laws stand behind them.

In order to go further in the process we are interested in, we must study more complex examples, namely where more that one monad is involved [Jones and Duponcheel1993]. There are functions that require more than one computational effect to model their behavior. The usual process to obtain a monad with several effects is to take a base monad and "enrich" it with more properties. The new generated monad is called a *monad transformer* [Liang et al.1995].

On the other hand, there are monadic properties and laws that have not been used in this paper, in which we have only used laws and properties relevant to the examples developed. In particular, we have applied new monadic properties related to the *either* and *sum* operators. It remains to be seen what monadic laws are related to the rest of the known operators, and what their role in our current approach is.

Our attention in this paper has been focused on the well known monad exception. The same analysis remains to be done on the other monads that are important for modelling the semantics of imperative programs, namely nondeterminism, interactive input and output, etc.

As pointed out in section 1, slicing techniques have been left aside to avoid additional difficulties. So, after getting a clear view on how to operate on monadic functions, we must retake the approach described in [Oliveira and Villavicencio2001, Villavicencio2003] and put slicing techniques back into the game.

5 Conclusions

In this paper we have developed some examples of program analysis by reverse specification in the presence of effects described monadically with the purpose of algebraically deriving abstract (point-free) descriptions of the original code. New laws and properties have been added to the repertoire of [Oliveira and Villavicencio2001, Villavicencio2003].

We have shown that the presence of side effects entails an important enrichment in our approach: the use of monads to describe and reason about effects.

However, we have not sufficiently identified generic schemata of monadic transformations. Therefore, the transformations we have applied are intuitive in nature. Although there are "cosmetic" transformations that can be automated, most of them cannot. However, having a clear goal to achieve in a transformation process is extremely important in selecting which rule to apply. Thus, we are heading to a semi-automatic refactoring process instead of a fully automated one. Alternatively, this could be an opportunity for incorporating refactoring tools in **HASKELL** code such as eg. **HaRe** [Li et al.2003] the monadic transformations required by the RPC process which are not supported therein.

References

[Barendregt1984] Barendregt, H. (1984) , *The Lambda Calculus: Its Syntax and Semantics*, North-Holland, revised ed.

[Barendregt1997] Barendregt, H. (1997) , *The Lambda Calculus*, North-Holland, 2 edition

[Benton et al.2000] Benton, N., Hughes, J., and Moggi, E. (2000) , In *APPSEM:International Summer School on Applied Semantics*, Vol. 2395 of *LNCS*, Springer-Verlag

[Cunha and Pinto2004] Cunha, A. and Pinto, J. S. (2004) , In *2nd. APPSEM II Workshop*, pp. 178–179

[de Moor and Gibbons2000] de Moor, O. and Gibbons, J. (2000) , In *AMAST: 8th International Conference on Algebraic Methodology and Software Technology*, Vol. 1816 of *LNCS*, pp. 371–390, Springer-Verlag

[Erkök2002] Erkök, L. (2002) , *Ph.D. thesis*, OGI School of Science and Engineering, Oregon Health and Science University

[Erkök and Launchbury2002] Erkök, L. and Launchbury, J. (2002) , In *Haskell Workshop 2002*

[Erwig and Ren2004] Erwig, M. and Ren, D. (2004) , *Science of Computer Programming* **52(1-3)**, 101

[Filinski1996] Filinski, A. (1996) , *Ph.D. thesis*, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA

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

[Jones and Duponcheel1993] Jones, M. and Duponcheel, L. (1993) , *Composing Monads*, Technical Report YALEU/DCS/RR-1004, New Haven, Connecticut, USA: Dept. of Computer Science, Yale University

[Li et al.2003] Li, H., Reinke, C., and Thompson, S. (2003) , In *ACM SIGPLAN 2003 Haskell Workshop*. (J. Jeuring ed.), pp. 27–38, ACM

[Liang et al.1995] Liang, S., Hudak, P., and Jones, M. (1995) , In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, CA: ACM Press, New York, 1995

[Meijer and Jeuring1995] Meijer, E. and Jeuring, J. (1995) , In *Tutorial Text 1st Int. Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, 24–30 May 1995*. (J. Jeuring and E. Meijer eds.), Vol. 925, pp. 228–266, Berlin: Springer-Verlag

[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

[Moggi and Sabry2004] Moggi, E. and Sabry, A. (2004) , *Theoretical Informatics and Applications*, To appear

[Oliveira1999] Oliveira, J. (1999) , *An Introduction to Point-free Programming*, Departamento de Informática, Universidade do Minho. 37p., chapter of book in preparation

[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

[Wadler1990] Wadler, P. (1990) , In *Conference on Lisp and Functional Programming*, pp. 61–78

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