Object Oriented Programming with Monadic Mealy Machines

Victor Cacciari Miraldo

`victor.cacciari 'at' gmail.com`

Techn. Report TR-HASLab:04:2014

Oct. 2014

HASLab - High-Assurance Software Laboratory
Universidade do Minho
Campus de Gualtar – Braga – Portugal
http://haslab.di.uminho.pt

TR-HASLab:04:2014
Object Oriented Programming with Monadic Mealy Machines
by Victor Cacciari Miraldo

Abstract

Object Orientation is a well established programming paradigm. The idea of having an entity whose *identity* and *memory* persist over time and can be interacted with through an *interface* is very appealing to programming large-scale software. Functional specifications, on the other hand, are much cleaner and allow us to apply a number of reasoning techniques to prove properties of a given specification. Ideally, we would like to *think* using objects and *calculate* with their functional specification.

It turns out that there are connections between both paradigms. This technical report emerges from the ideas behind a so-called *objectification* law. We give semantics for a toy object oriented language using software components in their functional form and showing that we can objectify software components and be able to trace their different interactions. We also address, albeit superficially, the connection between the behaviour monad and different language constructs that such monad corresponds to. A proof-of-concept compiler that outputs Haskell code, following the semantics presented here, was also developed as a basis for this research.

Object Oriented Programming with Monadic Mealy Machines

Victor Cacciari Miraldo
victor.cacciari 'at' gmail.com

University of Minho

Abstract. Object Orientation is a well established programming paradigm. The idea of having an entity whose *identity* and *memory* persist over time and can be interacted with through an *interface* is very appealing to programming large-scale software. Functional specifications, on the other hand, are much cleaner and allow us to apply a number of reasoning techniques to prove properties of a given specification. Ideally, we would like to *think* using objects and *calculate* with their functional specification.
It turns out that there are connections between both paradigms. This technical report emerges from the ideas behind a so-called *objectification* law. We give semantics for a toy object oriented language using software components in their functional form and showing that we can objectify software components and be able to trace their different interactions. We also address, albeit superficially, the connection between the behaviour monad and different language constructs that such monad corresponds to. A proof-of-concept compiler that outputs Haskell code, following the semantics presented here, was also developed as a basis for this research.

1 Introduction

Component Calculi, as introduced in [1], provides a coalgebraic approach to software engineering, where these so called components are coalgebras of a given category. This report is concerned with the semantics of a language in terms of components, and emerges as a continuation of [4], in terms of the expressiveness of the Monadic Mealy Machines formalism explained in the aforementioned paper.

The central idea of this report lies in theorem 1, where combinators $;$ and $\oplus$ represent sequential composition and choice, respectively.

Theorem 1 (Objectification exchange law). *Let m_1, m_2, n_1 and n_2 be Mealy Machines, assuming that they are suitably typed for the following construction, then*

$$(m_1 \oplus m_2) \; ; \; (n_1 \oplus n_2) = (m_1 \; ; \; n_1) \oplus (m_2 \; ; \; n_2)$$

On the left-hand side we can see two two method choices linked together, sequentially. Whereas on the right-hand side we see a choice between two sequential executions. The left-hand side is a *bottom-up* approach, where we put together two existing components using wires and compositions, contrasting with the *top-down* version, where we specify the functionality of the whole. Let us illustrate this with a simple example.

We will build a folder that provides two buttons, one for turning pages left and the other for turning them right. For this, we are going to use two communicating *stack* components.

Component oriented approach. Let us consider the following functions, from which we'll build a *stack* component:

$$\begin{aligned} &push :: (Monad\ \mathsf{F}) \Rightarrow ([p], p) \to \mathsf{F}\ ([p], 1) \\ &push\ (t, h) = \eta\ (h : t, ()) \end{aligned}$$

$$pop :: (Monad\ \mathsf{F}) \Rightarrow ([p], 1) \to \mathsf{F}\ ([p], p)$$

$$pop\ (l, ()) = \eta\ \$\ \langle tail, head\rangle\ l$$

$$stack :: (Strong\ \mathsf{F}) \Rightarrow ([p], 1+p) \to \mathsf{F}\ ([p], p+1)$$
$$stack = pop \oplus push$$

We can now (almost) construct our folder component. It consists in the external choice of two stacks, with a wiring separating the interface wires $1+1$ from the *private* ones, $p+p$, that will be used to feed back information.

$$\begin{aligned} preFolder :: &(Strong\ \mathsf{F}) \\ &\Rightarrow (([p],[p]),(1+1)+(p+p)) \\ &\to \mathsf{F}\ (([p],[p]),(1+1)+(p+p)) \\ preFolder = &\ stack \boxplus stack_{\{[[i_{11}\ ,i_{21}]\ ,[i_{12}\ ,i_{22}]]\to[[i_{21}\ ,i_{11}]\ ,[i_{22}\ ,i_{12}]]\}} \end{aligned}$$

We now need to specify how these folders will *talk*, that is, the result of popping the first stack must be pushed to the second and vice-versa, this is translated to:

$$\begin{aligned} connectFolder :: &(Strong\ \mathsf{F}) \\ &\Rightarrow (([p],[p]),(1+1)+(p+p)) \\ &\to \mathsf{F}\ (([p],[p]),(1+1)+(p+p)) \\ connectFolder = &\ (preFolder_{\{id\to \mathsf{s}_+ + \mathsf{s}_+\}})^{\curvearrowleft} \end{aligned}$$

Although $connectFolder$ is already a fully functional folder, the user can let it misbehave by giving, for instance, a $i_2\ (i_1\ p)$ as input. Thus we hide those buttons users shouldn't see:

$$\begin{aligned} folder :: &(Strong\ \mathsf{F}) \\ &\Rightarrow (([p],[p]),1+1) \\ &\to \mathsf{F}\ (([p],[p]),1+1) \\ folder = &\ connectFolder_{\{[i_{11}\ ,i_{12}]\to(id+\triangledown)\cdot \mathsf{a}^+\cdot(id+!)\}} \end{aligned}$$

our actions are, then:

$$tlBtn = i_2\ ()$$
$$trBtn = i_1\ ()$$

Method-oriented approach. The small example above gives evidence of the overhead introduced by wiring and interfacing components. Below we investigate how to build the *same* folder in a lighter, more functional flavor. For this we start the other way around (top-down rather than bottom-up) and specify what this folder should be able to do. Well, the only two actions it can do is turning pages, either to the left or to the right:

$$\begin{aligned} redlof :: &(Strong\ \mathsf{F}) \\ &\Rightarrow (([p],[p]),1+1) \\ &\to \mathsf{F}\ (([p],[p]),1+1) \\ redlof = &\ tr \oplus tl \end{aligned}$$

where

$$tl, tr :: (Strong\ \mathsf{F}) \Rightarrow (([p],[p]),1) \to \mathsf{F}\ (([p],[p]),1)$$
$$tr = pop\ \mathbf{;}\ push$$
$$tl = pop \overleftarrow{\mathbf{;}}\ push$$

capture the interaction between stacks in a method-driven way.

And now, we can use the same $tlBtn$ and $trBtn$ to control both our $folder$ and $redlof$ component. In fact, we can prove that they are the same component[3] and such proof uses the objectification theorem in it's core.

In this example we could see how to convert from a objectified version to a functional one, when we only have one communication channel (one variable being passed around, which is the page that is being turned). If we have more variables, a different wiring pattern arises. How should we deal with such pattern? How expresive the Mealy Machines formalism is? We seek to answer such questions in this report.

The structure of the document is as follows: section 2 will give a more formal introduction to components and clarify some naming conventions we'll use; In section 3 we explore the connection with object orientation; Section 4 introduces the language we'll be using to explain and define the semantics, that is given first informally in sections 5, 6 and 7, and formally in section 8. The appendix of the document contains examples and a simple usage manual for the compiler we developed.

2 Components, Machines and Objects

Before delving into anything more technical, we need to clarify some nomenclature problems. In [1], Barbosa defines a component as a (seeded) coalgebra for the functor:

$$\mathsf{T}^{\mathsf{B}} S = \mathsf{B}(S \times O)^I$$

where B is called the behavior monad, we require it to be strong[1]. This means that a given computation will produce a B-structure of a new state S and an output O. There are several possibilities for the behaviour monad, to name a few:

- *Pointed Machines*, the possibility of deadlock, failure or termination can be expressed by the maybe monad, $\mathsf{B} = \mathsf{Id} + 1$.
- *Error Codes*, analogous to the maybe monad, we can use a error type E and set $\mathsf{B} = \mathsf{Id} + E$ to allow special states with more information.
- *Nondeterminism* is modeled by the finite powerset monad, $\mathsf{B} = \mathcal{P}$.
- *Probabilistic Machines* live under the discrete sub-distribution monad $\mathsf{B} = \mathsf{Dist}$.

As in Haskell, in [5] we can see how we can combine different behaviour monads to achieve even more expressive machines. One restriction is that $\mathsf{B}\cdot\mathsf{B}'$ must be a strong monad. One of such combination that is being studied are the pointed probabilistic machines, where their behaviour is $\mathsf{Dist} \cdot (\mathsf{Id} + 1)$. The advantage of the component calculus is that we can work with almost arbitrary monads, and still use every definition.

For a more detailed explanation of which monads can be composed together, we refer the reader to [4]. Keeping it short, we can prove that if T is a strong monad and F the free monad[2] of a given functor, then, the lifting of F in the Kleisli Category of T is also a monad. The Kleisli category is important since, by definition, Monadic Mealy Machines compute results over *one* monad. With this technique, we're able to change the underlying category in which our Mealy machines are defined and keep (almost) every definition intact.

For the purpose of this document, we'll consider machines with the type:

$$(\mathit{Strong}\ \mathsf{B}) \Rightarrow (s, i) \to \mathsf{B}\ ((s, o) + e)$$

for an abstract B. Let's unroll this type and see what we're dealing with:

$$\begin{aligned}
& (s, i) \to \mathsf{B}\ ((s, o) + e) \\
\cong \quad & \{\ \text{swap +}\ \} \\
& (s, i) \to \mathsf{B}\ (e + (s, o)) \\
\cong \quad & \{\ \text{runErrorT}\ \} \\
& (s, i) \to \mathsf{ErrorT}\ e\ \mathsf{B}\ (s, o) \\
\cong \quad & \{\ \text{swap; curry}\ \} \\
& i \to s \to \mathsf{ErrorT}\ e\ \mathsf{B}\ (s, o) \\
\cong \quad & \{\ \text{runStateT}\ \} \\
& i \to \mathsf{StateT}\ s\ (\mathsf{ErrorT}\ e\ \mathsf{B})\ o
\end{aligned}$$

[1] A strong monad is a monad that has two natural transformations $\tau_r : \mathsf{B}X \times Y \to \mathsf{B}(X \times Y)$ and $\tau_l : X \times \mathsf{B}Y \to \mathsf{B}(X \times Y)$, they're already a well studied subject. More information can be found in [1]

[2] The free monad of a functor G is defined by $\mathsf{F}X = X + \mathsf{G}(\mathsf{F}X)$ with $\eta = \mathrm{in}_{\mathsf{G}}.i_1$ and $\mu = (\!|[id, \mathrm{in}_{\mathsf{G}}.i_2]|\!)$, where $(\!|\cdot|\!)$ denotes a catamorphism. It is simple to check that these definitions satisfy the monad laws.

Therefore, monadic Mealy Machines are just another view of a very common *transformer pattern*. The machines (or components) are presented as a state transition system that receives and sends pulses, while a *Haskeller* may view the component as a $\mathsf{StateT}\ s\ (\mathsf{ErrorT}\ e\ \mathsf{T})$ function, we may call this monad the imperative $\mathsf{ImpT}\ s\ e\ \mathsf{T}$ monad.

$$\mathbf{type}\ \mathsf{ImpT}\ s\ e\ \mathsf{T} = \mathsf{StateT}\ s\ (\mathsf{ErrorT}\ e\ \mathsf{T})$$

The ImpT monad confers some imperative-style constructs to a pure functional program. To the reader unfamiliar with such monads in Haskell, the StateT monad provides a $getState$ and $putState$ that can be used to manipulate what one would call global or class variables. Whereas the ErrorT provides a $throwError$ and $catchError$ functionality that works just like throwing and catching exceptions.

Our approach is different from what we can see in [2] where the authors define an interface functor that has explitic components for pure and impure computations of an object (computations that maintains or changes the internal state, respectively). Here, we take advantage of the state component already present in the Mealy Machines to model that. Pure computations turns out to be a simple lifting of a function.

Another difference is the degree of abstraction of both approaches. In [2] a generic functional language is considered and they *derive* an object for a specific catamorphism over a given datatype, where we merely provide an *encoding* of an object using Monadic Mealy Machines.

2.1 Selection of Combinators

In this section we'll introduce just enough combinators to be able to express normal programming tasks in terms of component algebra. Although this selection is small, it proved to be a minimal but expressive kernel. To keep things simple, some combinators will be introduced later, when the need for them arises.

Wiring. It is evident that we'll need to bring the whole pure-function machinery to our MMM universe, if we want to calculate anything. The $\cdot_{\{\cdot\to\cdot\}}$ and $\ulcorner\cdot\urcorner$ components serve this purpose.

$$\begin{aligned}
&\cdot_{\{\cdot\to\cdot\}} :: (Monad\ \mathsf{F}) \\
&\qquad \Rightarrow ((s,i) \to \mathsf{F}\ (s,o)) \\
&\qquad \to (i' \to i) \\
&\qquad \to (o \to o') \\
&\qquad \to (s,i') \to \mathsf{F}\ (s,o') \\
&m_{\{fi\to fo\}} = \ulcorner fo \urcorner \bullet m \bullet \ulcorner fi \urcorner
\end{aligned}$$

where $\ulcorner\cdot\urcorner$ lifts a ordinary function to a Mealy machine:

$$\begin{aligned}
&\ulcorner\cdot\urcorner :: (Monad\ m) \Rightarrow (a \to b) \to (s,a) \to m\ (s,b) \\
&\ulcorner f \urcorner = \eta \cdot (id \times f)
\end{aligned}$$

Sequential composition. State extension is essential to defining two forms of sequential machine composition, either forward composition (which sends data from the left component to right one)

$$\begin{aligned}
&\cdot\ ;\ \cdot :: (Strong\ \mathsf{F}) \Rightarrow \\
&\qquad ((s,i) \to \mathsf{F}\ (s,o)) \to \\
&\qquad ((r,o) \to \mathsf{F}\ (r,k)) \to \\
&\qquad ((s,r),i) \to \mathsf{F}\ ((s,r),k) \\
&p\ ;\ q = (extl\ q) \bullet (extr\ p)
\end{aligned}$$

or backwards sequential composition (which sends data from right to left):

$$
\begin{aligned}
&\cdot \overleftarrow{;} \cdot :: (\mathit{Strong}\ \mathsf{F}) \Rightarrow \\
&\qquad ((s, i) \to \mathsf{F}\ (s, o)) \to \\
&\qquad ((r, o) \to \mathsf{F}\ (r, k)) \to \\
&\qquad ((r, s), i) \to \mathsf{F}\ ((r, s), k) \\
&p \overleftarrow{;} q = (\mathit{extr}\ p) \bullet (\mathit{extl}\ q)
\end{aligned}
$$

Where extr and extl are state extensions to the right and to the left:

$$
\begin{aligned}
&\mathit{extr} :: (\mathit{Strong}\ \mathsf{F}) \Rightarrow \\
&\quad ((s, i) \to \mathsf{F}\ (s, o)) \\
&\quad ((s, r), i) \to \mathsf{F}\ ((s, r), o) \\
&\mathit{extr}\ p = \mathsf{F}\ \mathsf{xr} \cdot \tau_r \cdot (p \times \mathit{id}) \cdot \mathsf{xr}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{extl} :: (\mathit{Strong}\ \mathsf{F}) \Rightarrow \\
&\quad ((s, i) \to \mathsf{F}\ (s, o)) \\
&\quad ((r, s), i) \to \mathsf{F}\ ((r, s), o) \\
&\mathit{extl}\ q = \mathsf{F}\ \mathsf{a}^\circ \cdot \tau_l \cdot (\mathit{id} \times q) \cdot \mathsf{a}
\end{aligned}
$$

We are going to need a more general version of state extension, which is introduced in section 5. Although more general, we prove that in the simplest programming pattern, it reduces to the sequential composition.

Sum and Choice. A central aspect to programming languages is the ability to choose a method or procedure to run, this way we can write non-linear code. The $\cdot \oplus \cdot$ combinator is, for components, what methods are for classes. Given two components p and q, $p \oplus q$ is the component that runs either p or q over the *same state*.

$$
\begin{aligned}
&\cdot \oplus \cdot :: (\mathit{Monad}\ \mathsf{F}) \\
&\quad \Rightarrow ((s, i) \to \mathsf{F}\ (s, o)) \\
&\quad \to ((s, t) \to \mathsf{F}\ (s, u)) \\
&\quad \to (s, i + t) \to \mathsf{F}\ (s, o + u) \\
&m \oplus n. = \mathsf{F}\ (\mathsf{dr}^\circ) \cdot \mathit{cozip} \cdot (m + n.) \cdot \mathsf{dr}
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathit{cozip} :: (\mathit{Functor}\ \mathsf{F}) \Rightarrow (\mathsf{F}\ a) + (\mathsf{F}\ b) \to \mathsf{F}\ (a + b) \\
&\mathit{cozip} = [\mathsf{F}\ i_1\ , \mathsf{F}\ i_2]
\end{aligned}
$$

Taking advantage of the state extensions, we can also extend the shared-state sum to run each component in it's own, separate state.

$$
\begin{aligned}
&\cdot \boxplus \cdot :: (\mathit{Strong}\ \mathsf{F}) \Rightarrow \\
&\qquad ((s, i) \to \mathsf{F}\ (s, o)) \to \\
&\qquad ((t, j) \to \mathsf{F}\ (t, r)) \to \\
&\qquad ((s, t), i + j) \to \mathsf{F}\ ((s, t), o + r) \\
&p \boxplus q = \mathit{extr}\ p \oplus \mathit{extl}\ q
\end{aligned}
$$

Split. Besides sums, we'll also need a form of product to be able to couple inputs and outputs together. Although named *split*, this combinator is *not* a categorical split, in fact, in an arbitrary $\langle p, q \rangle$ we don't even have cancellation laws (in the particular case when one component is a pure function the cancellation laws hold, as it can be seen in appendix A).

$$
\begin{aligned}
&\langle \cdot, \cdot \rangle :: (\mathit{Strong}\ \mathsf{F}) \Rightarrow \\
&\quad ((s, i) \to \mathsf{F}\ (s, j)) \to \\
&\quad ((s, i) \to \mathsf{F}\ (s, k)) \to \\
&\quad (s, i) \to \mathsf{F}\ (s, (j, k)) \\
&\langle p, q \rangle = \mathsf{F}\ ((\mathit{id} \times \mathsf{s}) \cdot \mathsf{a}) \cdot (\tau_r \cdot (q \times \mathit{id}) \cdot \mathsf{xr}) \bullet (\tau_r \cdot (p \times \mathit{id}) \cdot \langle \mathit{id}, \pi_2 \rangle)
\end{aligned}
$$

Identity. Having a neutral element in our MMM toolbox will prove to be extremely useful, it is, in fact, a very simple machine:

$$copy :: (Strong\ \mathsf{F}) \Rightarrow (s, i) \to \mathsf{F}\ (s, o)$$
$$copy = \ulcorner id \urcorner$$

Now that we have a simple, yet powerful, set of combinators, let us build the intuitive connections between them and the world of objects.

3 Relationship with OOP

Starting by the very definition of object, in the OOP setting: it is a collection of methods that work over an *encapsulated* state, that is, the object itself is *the* state (in $C++$, the keyword `this` is a pointer to the state structure). The methods just change the object and the type system guarantees that everything works fine. We can formalize this notion using component calculus, assuming that our state is S and we have a collection $m_1, \cdots, m_k$ of methods that operate over it, each with type $S \times I_i \to \mathsf{F}(S \times O_i)$, $0 < i \leqslant k$. We then define the *object* M as:

$$M = \bigoplus_{i \leqslant k} m_i \tag{1}$$

If we were to write, for instance, Java code, to run m_j for some $j \leqslant k$ we would use the method selection constructor, the result would be something like $M.m_j$. In the component-oriented mindset, one would use wires to provide access only to the j-th[3] component of our object: $M_{\{\mathsf{in}_j \to id\}}$, where in_j has type $I_j \to \sum_{i \leqslant k} I_i$, for $j \leqslant k$. In other words, we're running only m_j and wiring its output with in_j.

Theorem 2 (Method Selection). *Selecting a method from a object is the same as running this method and* masking *its output as the object's output.*

$$M_{\{\mathsf{in}_j \to id\}} = m_{j\,\{id \to \mathsf{in}_j\}}$$

Now we know at least what basic language features we're able to model using MMM's. More complex topics such as inheritance, error-handling and probabilistic methods shall be addressed further down the road.

If we take a look at the *echoice* combinator, we can see it provides some sort of class inheritance. It respects the object definition, that is, if A and B are coproducts, then so is $A + B$. And it allows us to add more state variables to the new class. It is not proper inheritance since the subclass have no access to the superclass state, for that reason we'll not deal with such topics here, and this subject is addressed as future work.

4 A Minimal Language

For the moment, we'll only introduce a minimal toy language. The features are just the default features one finds in any programming language plus the notion of object, that is, an encapsulated state and some syntax-sugar to implicitly pass this state to our methods.

A Class $C = (V, M)$ is defined by variables $v \in \mathcal{V}(C)$ and methods $m \in \mathcal{M}(C)$, for illustration purposes we'll consider a minimal OOP language to illustrate the translation into (monadic) Mealy Machines. The classical Stack example follows:

[3] Although every definition of binary coproducts is extensible to finitary sums (coproducts), in order to implement these notions in Haskell, for instance, we would need a suitable definition for this in_j. Assume the coproducts are left associative and define:

$$\mathsf{in}_j = i_1^{(k-j)} \cdot i_2^{(\delta j)} \qquad \text{where } \delta j = \begin{cases} 0, j = 1, \\ 1, j > 1 \end{cases}$$

```
class Stack of p
    var st :: list of p;

    Stack(s :: list of p) {
      st = s;
    }

    method pop() :: p {
      i = head st;
      st = tail st;
      return i;
    }

    method push(a :: p) :: void {
      st = p @ st; // @ is the cons operator.
    }
```

You probably noticed that we use a basic `list of a` type, which works just like the Haskell $[a]$. In fact, that's the only structured type we'll provide built-in, equipped with the functions: *head*, *tail*, *cons* and *length*.

As discussed in section 3, the methods are not part of the object itself, but ways to interact with it. Therefore, the inhabitants of class Stack are the lists $[p]$, and the only possible interactions we can have with them are through pop and $push$. Given an arbitrary class C, the inhabitants of C are those with type

$$type(C) = \prod \{t \mid \texttt{var x :: t} \in C\}$$

For practical reasons, we'll consider this product (and every other, unless mentioned) to be in it's left-associative form. And we also define a family of projections for each `var x :: t` in $\mathcal{V}(C)$ by $\pi_{\texttt{x}} : type(C) \to t$, which is just a n-ary projection in the component corresponding to where x is *stored*.

4.1 Code Transformations

We cannot translate the code *as is*, since it is too general. We need to gather some assumptions about it before attempting to write it as monadic Mealy machines. In this section we'll provide a brief explanation of the transformations applied to the code and how do they help us achieve a successful translation.

Statement Purification. Impure and pure statements are separated. Impure statements are those that access (set, read or modify) the object's state. If we can assume that a statement is either pure or impure, we can immediately know if it's going to be just a function lifted to a machine (pure machine) or not.

```
x = this.v.m(2*y)          ⟹ stmt−purify ⟹      aux1 = this.v.m(2*y);
    + 6*this.v.parm;                             aux2 = this.v.parm;
                                                 x = aux + 6*aux2;
```

Expression Linearization. A linear expression is one that has, at most, one operator. We transform every expression into it's corresponding list of linear sub-expressions. This step is crucial to variable management. We now know that a given statement either modifies the state or applies an operation to, at most, two other variables.

```
aux1 = this.v.m(2*y);                          aux3 = 2*y;
aux2 = this.v.parm;        ⟹ exp−lin ⟹         aux1 = this.v.m(aux3);
x = aux + 6*aux2;                              aux2 = this.v.parm;
                                               aux4 = 6*aux2;
                                               x = aux1 + aux4;
```

If-Completion. This transformation is explained in more detail in section 7. In short, we garbage-collect the variables that are created in one branch only, this transformation allows us to unify the output type of both branches.

Explicit Getters and Setters. After having linear expressions and either pure or impure statements (not mixed ones), we lift the impure assignments to special constructors. This step is not really necessary but it makes the implementation easier.

```
aux3 = 2*y;                                    aux3 = 2*y;
aux1 = this.v.m(aux3);                         aux1 = this.v.m(aux3);
aux2 = this.v.parm;          ⟹ explicit−getset ⟹  GET(this.v.parm, aux2);
aux4 = 6*aux2;                                 aux4 = 6*aux2;
x = aux1 + aux4;                               x = aux1 + aux4;
```

5 Managing State, or, Class Variables

Programming is about setting and retrieving values, the variables are just labels. Just like normal imperative programming and RAM memory, we need to store our values into some ordered *containers*, in our case, specially when we want to generate Haskell code to run our objects, our container is of type $type(C)$, and the address of the variables are the position they occupy in the container. For instance, if we have a class C with four variables: $x :: a,\ y :: b,\ w :: c$ and $z :: a$, declared in this order. Then $type(C) = ((a \times b) \times c) \times a$, and the address of x is 1, whereas the address of w is 3.
Let's consider global (or state) variables for now, we'll discuss the handling of locals later. Retrieving x value is very simple. Let's consider the following component, that outputs it's state:

$$get :: (Monad\ \mathsf{F}) \Rightarrow (s, ()) \to \mathsf{F}\ (s, s)$$
$$get = \eta \cdot \langle \pi_1, \pi_1 \rangle$$

Whenever we need x value, we can use $\ulcorner \pi_x \urcorner \bullet get$
Setting it's value is analogous, but the setter depends on the type of the object. Consider that we have a class C with $type(C) = (a \times b) \times c$ and we wish to set the second variable (index 2, type b). Our *setter* would be:

$$set\_C\_b :: (Monad\ \mathsf{F}) \Rightarrow (((a, b), c), b) \to \mathsf{F}\ (((a, b), c), ())$$
$$set\_C\_b = \eta \cdot \langle \langle \langle (\pi_1 \cdot \pi_1 \cdot \pi_1), \pi_2 \rangle, (\pi_1 \cdot \pi_2) \rangle, ! \rangle$$

As expected, working with class variables is very close to working with the state monad. If we use no other classes inside a class C, everything works fine. Some care must be taken, though, to call methods over other classes. We need to extend the method state with $extl$ and $extr$ until it is compatible with C's state.
To illustrate this, let's consider a folder:

```
class Folder of p
    var stL :: Stack of p;
    var stR :: Stack of p;

    Folder(s :: Stack of p) {
      stL = s;
      stR = new Stack([]);
    }

    method tr() : void {
      page = stL.pop();
•     stR.push(page);
    }

    ...
```

The type of Folder is $type(stL) \times type(stR) \equiv type(Stack) \times type(Stack) \equiv [p] \times [p]$. Yet, in line •, we're calling $push$ on one of it's variables. We can't pass a Folder to push because it does not typecheck and we can't separate the two stacks from the folder as they are *the* folder we want to manipulate, but we can accommodate $push$ to just do nothing with the other stack:

$$extl\ push :: (Monad\ \mathsf{F}) \Rightarrow ((a, [p]), p) \to \mathsf{F}\ ((a, [p]), ())$$

Let C be a class, $x \in \mathcal{V}(C)$ a class variable, i the address of x and $\#C$ the dimension of the state space. We then define the generalized state extension of x in C by:

$$\mathsf{ext}_x^C = extl^{\#C-i}.extr^{\delta(i)}$$

Mind the type of ext_x^C, and note that it receives a mealy machine and returns a mealy machine. in fact, given a machine m, the machine $\mathsf{ext}_x^C\, m$ will run m on C's state space, but will only affect the x component.

6 Wiring Up, or, Local Variables

In general, when we encode a given component as a monadic Mealy Machine, we use wires to pass values from a subcomponent to another. In the technical report [3] we work the folder example and cite this paper for further information, since the generalization is not so straight-forward. The difficulty to provide a simple, concise, algorithm for encoding wires lies in the fact that there are infinite possible wiring patterns. However, with the help of some additional combinators we're able to mimic memory behaviour in our machine's input type. The main idea is simple: the input type has the form $I_1 \times I_2$. I_1 is a product corresponding to the variables that are going to be used in the next statement, we call this the focus of the product, I_2 is a product of the unchanged variables.
It's worthwhile to mention that if we were translating to a language that had support for heterogeneous lists ($l :: [forall\ \ a \cdot a]$) with run-time type-casting operations, lists would be a much simpler choice for storing local variables. Since we're translating to Haskell, we have to stick with (binary) products.

$$
\begin{aligned}
&runm :: (Strong\ m) \Rightarrow \\
&\qquad ((s, i) \to m(s, o)) \to \\
&\qquad ((s, (i, b)) \to m(s, (i, (b, o)))) \\
&runm\ p = \ulcorner \mathsf{a} \urcorner \bullet \langle copy, (p \bullet \ulcorner \pi_1 \urcorner) \rangle
\end{aligned}
$$

$$
\begin{aligned}
&runm\_ :: (Strong\ m) \Rightarrow \\
&\qquad ((s, i) \to m(s, o)) \to \\
&\qquad ((s, (i, b)) \to m(s, (i, b))) \\
&runm\_\ p = \ulcorner \pi_1 \cdot \mathsf{a}^{\circ} \urcorner \bullet runm\ p
\end{aligned}
$$

The above combinators are two possible ways of running a *statement*. It either assigns a value to a variable, that is, creates a new component in the unchanged part of our input, or we only need it's side effects. Before running each machine, we need to *select* the needed variables and, if it assigns a new value to an existing variable, we need to *rewrite* such variable.

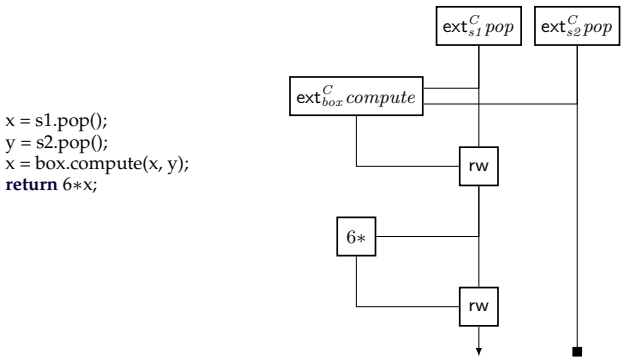

Fig. 1. Graphical Representation

Note how the output of each new step is it's input and a new value *appended* to it. That is, in the Kleisli Category of F we have the following diagram (Let's assume that the types of x and y are X and Y, respectively).

$$
\begin{aligned}
S \times I &\xrightarrow{\ulcorner pwnil \urcorner} S \times (1 \times I) \\
&\xrightarrow{runm\ (\mathsf{ext}^C_{s1}\, pop)} S \times (1 \times (I \times X)) \\
&\xrightarrow{\mathsf{select}^{[\,]}} S \times (1 \times (I \times X)) \\
&\xrightarrow{runm\ (\mathsf{ext}^C_{s2}\, pop)} S \times (1 \times ((I \times X) \times Y)) \\
&\xrightarrow{\mathsf{select}^{[x,y]}} S \times ((X \times Y) \times (1 \times I)) \\
&\xrightarrow{runm\ (\mathsf{ext}^C_{box}\, compute)} S \times ((X \times Y) \times ((1 \times I) \times X)) \\
&\xrightarrow{\mathsf{rw}\ x} S \times (((X \times Y) \times 1) \times I) \\
&\xrightarrow{\mathsf{select}^{[x]}} S \times (X \times ((Y \times 1) \times I)) \\
&\xrightarrow{runm\ \ulcorner \lambda a \rightarrow 6 * a \urcorner} S \times (X \times (((Y \times 1) \times I) \times X)) \\
&\xrightarrow{\mathsf{rw}\ x} S \times (((X \times Y) \times 1) \times I) \\
&\xrightarrow{\mathsf{select}^{[x]}} S \times (X \times ((Y \times 1) \times I)) \\
&\xrightarrow{\ulcorner pwnil^{\circ} \cdot \langle \pi_1, ! \rangle \urcorner} S \times X
\end{aligned}
$$

Where $pwnil$ is the $A \times 1 \cong A$ isomorphism.
So, we're using a fancy associativity isomorphism, called $\mathsf{select}^{\cdot}$, to push the variables we need to the first component then we run the desired machine, then we fix the output type with a projection after associativity. We join everything together using Kleisli compositions and the final result is a mealy machine, equivalent to the initial code.

6.1 Selecting and Rewriting

The $\mathsf{select}^{\cdot}$ and rw are meta-combinators. We can't encode them directly into Haskell since they require dependent types to be written. Yet we can see their types and discuss how to generate them on-the-fly when translating machines. Let V be a set of variables with types T_v for each $v \in V$ and $P \subseteq V$, then:

$$select^P_V :: \prod_{i \in V} T_i \to \prod_{j \in P} T_j \times \prod_{i \in V \setminus P} T_i$$

It's easy to see that $\mathsf{select}^{\cdot}$ is an isomorphism, it is just a generalization of the s isomorphism for finitary products. For practical matters, since Haskell does not support finitary products, we have to translate every finitary product into a binary one. This is easy to do, as it has been explained in the previous sections, but we gotta take special care with how we associate our products. Note that $\mathsf{select}^{\cdot}$ creates a *focus*, breaking the left-associativity constraint into a product of two left-associated products. It is important to note that $\mathsf{select}^{[\,]}$ has type $1 \times \prod_i T_i$, that is, selects a button (or a void variable). The rewrite operator has type:

$$rw^x_V :: \prod_{i \in V} T_i \times T_x \to \prod_{i \in V} T_i$$

And it can be easily implemented by (where x' denotes the new value of x. Note that $T_x = T_{x'}$ since our language is statically typed):

$$
\begin{aligned}
\prod_{i \in V} T_i \times T_{x'} &\xrightarrow{\mathsf{select}^{[x,x']}} (T_x \times T_{x'}) \times \prod_{i \in V \setminus \{x\}} T_i \\
&\xrightarrow{\pi_2 \times id} T_x \times \prod_{i \in V \setminus \{x\}} T_i \\
&\xrightarrow{\mathsf{a}^{\circ}} \prod_{i \in V} T_i
\end{aligned}
$$

Where a° is a left-associativity isomorphism generalized to finitary products.
Another useful house-keeping operator that can strongly influence the space complexity of the generated code is the garbage-collection, $\mathsf{collect}^{\cdot}$, meta combinator:

$$\mathsf{collect}^l = \prod_{i \in V} T_i \times T_{x'} \xrightarrow{\mathsf{select}^l} \prod_{i \in l} T_i \times \prod_{i \in V \setminus l} T_i \xrightarrow{\pi_2} \prod_{i \in V \setminus l} T_i$$

The lifting of $\mathsf{select}^{\cdot}$, $\mathsf{collect}^{\cdot}$ and rw to Mealy Machines is trivial. We may use them as functions and machines interchangeably. Distinction will be made only when the context is not clear.
A simple example with the meta-combinators translated to Haskell, is provided in figure 2. Although the code is not readable, it compiles and works correctly. We invite the not faint-of-heart to check the types of the $\mathsf{select}^{\cdot}$'s with the help of ghci in order to see this *focusing* heuristic working.

```
inc(i : int) : void {
  this.c = this.c + i;
}
```

$\xrightarrow{simplify}$

```
inc (i : int) : void {
  GET(this.c, p0);
  l2 = (p0 + i);
  SET(this.c, l2);
  return NIL;
}
```

$$\begin{aligned}
counterINC &:: (Strong\ \mathsf{F}) \Rightarrow (Integer, Integer) \to \mathsf{F}\ (Integer, ()) \\
counterINC &= \ulcorner \pi_1 \cdot _a4 \urcorner \\
&\bullet\ \ulcorner \pi_1 \cdot _a3 \urcorner \\
&\bullet\ runm_\ set_c_Counter \\
&\bullet\ \ulcorner _a2 \urcorner \\
&\bullet\ runm\ \ulcorner \lambda(_pure0, i) \to (_pure0 + i) \urcorner \\
&\bullet\ \ulcorner _a1 \urcorner \\
&\bullet\ runm\ get_c_Counter \\
&\bullet\ \ulcorner _a0 \urcorner \\
&\bullet\ runm\ get_c_Counter \\
&\bullet\ \ulcorner nilpw \urcorner \\
&\textbf{where} \\
&\quad _a0 = \langle \pi_1, \pi_2 \rangle \\
&\quad _a1 = \langle \langle (\pi_2 \cdot \pi_2), (\pi_1 \cdot \pi_2) \rangle, (id \cdot \pi_1) \rangle \\
&\quad _a2 = \langle (\pi_2 \cdot \pi_2), \langle \langle (\pi_1 \cdot \pi_1), (\pi_2 \cdot \pi_1) \rangle, (\pi_1 \cdot \pi_2) \rangle \rangle \\
&\quad _a3 = \langle (\pi_2 \cdot \pi_2), \langle \langle (id \cdot \pi_1), ((\pi_1 \cdot \pi_1) \cdot \pi_2) \rangle, ((\pi_2 \cdot \pi_1) \cdot \pi_2) \rangle \rangle \\
&\quad _a4 = \langle (id \cdot \pi_1), \langle \langle ((\pi_1 \cdot \pi_1) \cdot \pi_2), ((\pi_2 \cdot \pi_1) \cdot \pi_2) \rangle, (\pi_2 \cdot \pi_2) \rangle \rangle
\end{aligned}$$

Fig. 2. A counter's inc: from raw code to MMM

7 Branching

Everything seems to be working fine so far, but we still have no means to handle non-linear code. In this section we shall explain how branches can be translated into Mealy Machines.

In order to handle *if-then-else* constructs we need to apply some code transformations beforehand. In most imperative-style languages conditionals are regarded as statements, this makes them very handy for writing programs but it's also too general. To illustrate the problem we should take a look at the (modified) McCarthy conditional on Monadic Mealy Machines:

$$
\begin{aligned}
&\cdot \to \cdot, \cdot :: (Strong\ m) \Rightarrow \\
&\qquad (i \to \mathbb{B}) \to \\
&\qquad (((s, i) \to m(s, o))) \to \\
&\qquad (((s, i) \to m(s, o))) \to \\
&\qquad ((s, i) \to m(s, o)) \\
&c \to th, el = \ulcorner \triangledown \urcorner \bullet th \oplus el \bullet \ulcorner grd\ c \urcorner
\end{aligned}
$$

Note how both branches must have the same input and output type. Remember that the input type of our machines represent our memory, and store the variables that are in scope at a given moment. Therefore, in order to unify their types, they need to use the same variables. One alternative is to complete the branches by adding $\perp$ assignments and rely on lazy evaluation of Haskell to never evaluate them. Yet, we can do a better job by using our collect$^{\cdot}$ meta-combinator. Figure 3 illustrates this transformation. By collecting the variables that are used in only one of the branches we're able to unify their output types and save memory, by not accumulating variables that are not going to be used.

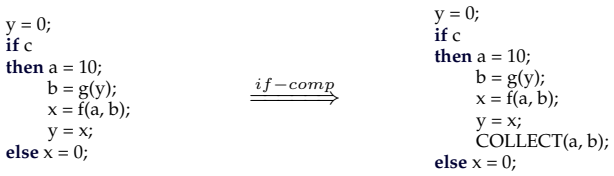

Fig. 3. If-completion transformation

And, with both branches being unifiable, we're able to use the $\cdot \to \cdot, \cdot$ combinator to give semantics to our conditional statements.

8 Semantics

In this section we'll provide a summary of the semantics discussed above, although not formally mentioned in the definition, we'll denote by V' the new variable set after performing a semantic translation step. Let C be a class:

$$[\![C]\!] = \bigoplus_{m \in \mathcal{M}(C)} [\![m]\!]_\emptyset$$

$$[\![func(x_1, \cdots, x_n) : t\{\ body\ \}]\!]_V = [\![body]\!]_{V \cup \{x_1, \cdots, x_n\}}$$

$$[\![x = f(x_1, \cdots, x_n)]\!]_V = \begin{cases} runm1\ \ulcorner f \urcorner,\ \#V = 1 \\ runm\ \ulcorner f \urcorner,\ otherwise. \end{cases} \bullet \mathsf{select}^{[x1..xn]}$$

$$[\![x = obj.m(x_1, \cdots, x_n)]\!]_V = \begin{cases} runm1\ (\mathsf{ext}^C_{obj}\, m),\ \#V = 1 \\ runm\ \ (\mathsf{ext}^C_{obj}\, m),\ otherwise. \end{cases} \bullet \mathsf{select}^{[x1..xn]}$$

$$[\![obj.m(x_1, \cdots, x_n)]\!]_V = runm\_\ (\mathsf{ext}^C_{obj}\, m) \bullet \mathsf{select}^{[x1..xn]}$$

$$[\![this.v = x]\!]_V = runm\_\ set\_C\_v \bullet \mathsf{select}^{[x]}$$

$$[\![x = this.v]\!]_V = runm\ (\ulcorner \pi_v \urcorner \bullet get) \bullet \mathsf{select}^{[]}$$

$$[\![COLLECT(l)]\!]_V = \mathsf{collect}^l, \mathsf{set}\ V = V \setminus l$$

$$[\![return\ x]\!]_V = \ulcorner \pi_1 \urcorner \bullet \mathsf{select}^{[x]}$$

$$[\![\mathbf{if}\ c\ \mathbf{then}\ a\ \mathbf{else}\ b]\!]_V = (\pi_c) \to [\![a]\!]_V, [\![b]\!]_V, \text{ where } \pi_c :: \prod_{v \in V} T_v \to T_c$$

$$[\![s; sts]\!]_V = [\![sts]\!]_{V'} \bullet [\![s]\!]_V$$

It's important to note that for the variable assignment statements, the variable set has to change to $V \cup \{x\}$.
In [3], we work out a example by hand, showing the equivalence of it's functional and objectified versions, yet, the semantics given above are somewhat far away from the concise representation we arrived at in the aforementioned report. This distance is due to the infinite number of wiring patterns a programmer can use, therefore we need a more general approach. Nevertheless, the sequential composition combinator arises as the trivial case of the semantics given above.
Let us consider a class C with $type(C) = S \times R$, where $o1$ and $o2$ are it's subobjects with types S and R respectively. Let's now consider the (intuitive) sequential composition pattern (which is exactly the turning of pages of the folder component):

```
function () : a {
  p = this.o1.m1();
  x = this.o2.m2(p);
  return x:
}
```

We have that (we'll omit the *this* keyword for readability):

$$\left[\!\!\left[\begin{array}{l} \texttt{p = o1.m1();} \\ \texttt{x = o2.m2(p);} \\ \texttt{\ \ return x;} \end{array} \right]\!\!\right]_\emptyset$$

$$= [\![\texttt{return x;}]\!]_{\{p\}} \bullet [\![\texttt{x = o2.m2(p);}]\!]_{\{p\}} \bullet [\![\texttt{p = o1.m1();}]\!]_\emptyset$$

$$= [\![\texttt{return x;}]\!]_{\{p\}} \bullet [\![\texttt{x = o2.m2(p);}]\!]_{\{p\}} \bullet runm1\ (extr\ m_1) \bullet \mathsf{select}^{[]}$$

$$= [\![\texttt{return x;}]\!]_{\{p\}} \bullet runm\ (extl\ m_2) \bullet \mathsf{select}^{[p]} \bullet runm1\ (extr\ m_1) \bullet \mathsf{select}^{[]}$$

$$= \ulcorner \pi_1 \urcorner \bullet \mathsf{select}^{[x]} \bullet runm\ (extl\ m_2) \bullet \mathsf{select}^{[p]} \bullet runm1\ (extr\ m_1) \bullet \mathsf{select}^{[]}$$

Remembering that $\mathsf{select}^{\cdot}$ is a meta-combinator, we need to translate it to it's corresponding associativity isomorphism. The first selection, $\mathsf{select}^{[]}$, is translated to id, since we already have a *button* coming from the void input type of `function`. The second one must have type $1 \times p \to p \times 1$, therefore a swap suffices. The last $\mathsf{select}^{[x]}$, corresponding to the statement `return x`, must have type $p \times (1 \times x) \to x \times (1 \times p)$, so it is $\mathsf{s} \cdot \mathsf{xr} \cdot \mathsf{s}$, with s and xr being the isomorphisms $A \times B \to B \times A$ and $(A \times B) \times C \to$

$(A \times C) \times B$ respectively.

$$
\begin{aligned}
& \ulcorner \pi_1 \urcorner \bullet \ulcorner \mathsf{s} \cdot \mathsf{xr} \cdot \mathsf{s} \urcorner \bullet \mathit{runm}\ (\mathit{extl}\ m_2) \bullet \ulcorner \mathsf{s} \urcorner \bullet \mathit{runm1}\ (\mathit{extr}\ m_1) \bullet \ulcorner \mathit{id} \urcorner \\
= \quad & \{ \text{ identity; } \mathit{runm1} \text{ def; } \ulcorner \cdot \urcorner \text{ fusion } \} \\
& \ulcorner \pi_1 \cdot \mathsf{s} \cdot \mathsf{xr} \cdot \mathsf{s} \urcorner \bullet \mathit{runm}\ (\mathit{extl}\ m_2) \bullet \ulcorner \mathsf{s} \urcorner \bullet \langle \mathit{copy}, (\mathit{extr}\ m_1) \rangle \\
= \quad & \{ \mathit{runm} \text{ def; } \ulcorner \cdot \urcorner \text{ fusion } \} \\
& \ulcorner \pi_1 \cdot \mathsf{s} \cdot \mathsf{xr} \cdot \mathsf{s} \cdot \mathsf{a} \urcorner \bullet \langle \mathit{copy}, (\mathit{extl}\ m_2 \bullet \ulcorner \pi_1 \urcorner) \rangle \bullet \ulcorner \mathsf{s} \urcorner \bullet \langle \mathit{copy}, (\mathit{extr}\ m_1) \rangle \\
= \quad & \{ \pi_1 \cdot \mathsf{s} \cdot \mathsf{xr} \cdot \mathsf{s} \cdot \mathsf{a} = \pi_2 \} \\
& \ulcorner \pi_2 \urcorner \bullet \langle \mathit{copy}, (\mathit{extl}\ m_2 \bullet \ulcorner \pi_1 \urcorner) \rangle \bullet \ulcorner \mathsf{s} \urcorner \bullet \langle \mathit{copy}, (\mathit{extr}\ m_1) \rangle \\
= \quad & \{ \cdot_{\{\cdot \to \cdot\}} \text{ def; } \mathit{copy} = \ulcorner \mathit{id} \urcorner \} \\
& \langle \ulcorner \mathit{id} \urcorner, (\mathit{extl}\ m_2 \bullet \ulcorner \pi_1 \urcorner) \rangle_{\{\mathit{id} \to \pi_2\}} \bullet \ulcorner \mathsf{s} \urcorner \bullet \langle \mathit{copy}, (\mathit{extr}\ m_1) \rangle \\
= \quad & \{ \text{ Eq A.3 } \} \\
& \mathit{extl}\ m_2 \bullet \ulcorner \pi_1 \urcorner \bullet \ulcorner \mathsf{s} \urcorner \bullet \langle \mathit{copy}, (\mathit{extr}\ m_1) \rangle \\
= \quad & \{ \ulcorner \cdot \urcorner \text{ fusion; } \pi_1 \cdot \mathsf{s} = \pi_2 \} \\
& \mathit{extl}\ m_2 \bullet \ulcorner \pi_2 \urcorner \bullet \langle \mathit{copy}, (\mathit{extr}\ m_1) \rangle \\
= \quad & \{ \cdot_{\{\cdot \to \cdot\}} \text{ def; } \mathit{copy} = \ulcorner \mathit{id} \urcorner \text{; Eq A.3 } \} \\
& \mathit{extl}\ m_2 \bullet \mathit{extr}\ m_1 \\
= \quad & \{ \cdot \ ; \ \cdot \text{ def } \} \\
& m_1 \ ; \ m_2 \\
& \square
\end{aligned}
$$

At this point, we have all the tools we need to translate a simple language with standard features into Monadic Mealy machines. The thing is that we didn't even touch the monadic part yet. In the next section we'll give some thougths on how one could use monads to add more (realistic) features to the language.

9 Exploiting the Behavior Monad

In section 2 we talked about how the behavior monad B added a lot of expressivity. Recall the example we took, with $\mathsf{B} = \mathsf{B}'.(+e)$ (note that the state-monadic component is already part of our Mealy Machines, by construction), or, in Haskell:

$$
\begin{aligned}
& (\mathit{Strong}\ \mathsf{B}) \Rightarrow (s, i) \to \mathsf{B}\ ((s, o) + e) \\
= \quad & \{ \ \} \\
& i \to \mathsf{StateT}\ s\ (\mathsf{ErrorT}\ e\ \mathsf{B})\ o
\end{aligned}
$$

Which means that we can easily add error-handling functionality to our language. In fact, if we look at the declaration of $\mathit{MonadError}\ e\ m$ we have:

$$
\begin{aligned}
& \textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadError}\ e\ m \mid m \to e\ \textbf{where} \\
& \quad \mathit{throwError} :: e \to m\ a \\
& \quad \mathit{catchError} :: m\ a \to (e \to m\ a) \to m\ a
\end{aligned}
$$

Looking at both available functions, they are not Mealy Machines, but they can be translated into one. First we need a (parametrized) machine that always throws an $\mathit{Exception}$[4]:

[4] In the declaration of $\mathit{MonadError}\ e\ m$, m uniquely determines e for each instance, for everything to work nicely inside Haskell, we also need to declare an instance $\mathit{Error}\ e$. This is easy though. A static analysis could determine the labels of each thrown exception in our code and combine them in a datatype called $\mathit{Exception}$, that would be an instance of Error.

$$
\begin{aligned}
&mthrow :: (Strong\ \mathsf{F}) \Rightarrow \\
&\qquad Exception \rightarrow \\
&\qquad (s, i) \rightarrow (\mathsf{ErrorT}\ Exception\ \mathsf{F})\ (s, o) \\
&mthrow\ e = \underline{(throwError\ e)}
\end{aligned}
$$

And, one that catches possible exceptions:

$$
\begin{aligned}
&mcatch :: (Strong\ \mathsf{F}) \Rightarrow \\
&\qquad ((s, i) \rightarrow (\mathsf{ErrorT}\ Exception\ \mathsf{F})\ (s, o)) \rightarrow \\
&\qquad ((s, Exception) \rightarrow (\mathsf{ErrorT}\ Exception\ \mathsf{F})\ (s, o)) \rightarrow \\
&\qquad (s, i) \rightarrow (\mathsf{ErrorT}\ Exception\ \mathsf{F})\ (s, o) \\
&mcatch\ p\ handle\ (s, i) = p\ (s, i)\ `catchError`\ (q \cdot \langle \underline{s}, id \rangle)
\end{aligned}
$$

Now we're able to add two language constructs, a *throw* and *catch* statements that behave just like expected. Their translation into MMM is just $mthrow$ and $mcatch$. We need to be careful and add $\mathsf{ErrorT}\ Exception\ \mathsf{F}$ instead of only F in the type-sig of every method and lift the pure F parts to ErrorT.
Besides error handling, we could add probabilistic extensions. For instance, if we also consider $DistT\ \mathsf{B}$, the monad of probability (sub-)distributions, in our chain of monads, we would also be able to add a keyword to model function failure, for instance:

```
increment( i : int ) : int
  with 80% {
    return i+1;
  }
  with 15% {
    return i;
  }
  with 5% {
    return 0;
  }
```

And, using the $DistT\ \mathsf{F}$ monad machinery, we could still implement this as a MMM, of course, taking care to correctly lift the F functions properly and adding the correct type signature to the proper places.

10 Conclusions and Future Work

The translation of Object Oriented Programming into Monadic Mealy Machines is a first step in a proof of a general objectification theorem. By writing wired-MMMs as Classes and translating them to MMMs that use no wires we can see that a pattern starts to unfold. It is evident that the generated code could be simplified, in fact, this is an interesting line of future work. Can we simplify it enough so that we can witness the exchange law?
Another very interesting question is how can we use the Abstract Strong Monad present in a MMM type to add semantics to different language constructs. Exception Handling is the most straightforward of these, as we already showed. What if we start to consider more combinators into our translation, what is the meaning of the rest of the component combinators in terms of object-oriented code? In fact, we would like to implement the monadic layer into our compiler and investigate further practical monads and see what language constructs they correspond to. One very positive, immediate, result of adding failure probabilities to our language is the ability to compute the probability of an overall failure given the probability of the parts failing.
In this document we provided a semantic of a toy-language into MMMs and shown a proof-of-concept tool developed to illustrate the algorithm. Now, we're able to formally express the connection between programming with objects and components.

References

1. Luis Soares Barbosa. *Components as Coalgebras*. PhD thesis, University of Minho, 2001.

2. Ralf Lämmel and Ondrej Rypacek. The expression lemma. In Philippe Audebaud and Christine Paulin-Mohring, editors, *Mathematics of Program Construction*, volume 5133 of *Lecture Notes in Computer Science*, pages 193–219. Springer Berlin Heidelberg, 2008.
3. J.N. Oliveira and V.C. Miraldo. Dividing machines. Technical report, 2014. (in preparation).
4. J.N. Oliveira and V.C. Miraldo. "keep definition, change category" — a lemma for state-based system calculi, 2014. Journal paper (submitted).
5. José N. Oliveira. Preparing relational algebra for "just good enough" hardware. In Peter Höfner, Peter Jipsen, Wolfram Kahl, and Martin Eric Müller, editors, *Relational and Algebraic Methods in Computer Science*, volume 8428 of *Lecture Notes in Computer Science*, pages 119–138. Springer International Publishing, 2014.

A Relevant Proofs

$\langle\cdot,\cdot\rangle$-*cancel* As we stated before, $\langle p, q\rangle$ is not isomorphic to $\langle q, p\rangle$, since the order of the side effects and state changes matters. Formally, $\langle\cdot,\cdot\rangle$ is *not* a categorical product. Nevertheless, if one of it's components is pure (that is, does not change the state nor has any side effect), we can prove some sort of cancelation properties about it:

$$(\langle p, \ulcorner f \urcorner\rangle)_{\{id\to\pi_1\}} = p \tag{2}$$
$$(\langle \ulcorner f \urcorner, q\rangle)_{\{id\to\pi_2\}} = q \tag{3}$$

We shall prove equation 2, the other one is analogous.

$$\begin{aligned}
& (\langle p, \ulcorner f \urcorner\rangle)_{\{id\to\pi_1\}} \\
= & \quad \{\ \cdot_{\{\cdot\to\cdot\}}\ \text{def};\ \langle\cdot,\cdot\rangle\ \text{def}\ \} \\
& \ulcorner \pi_1 \urcorner \bullet \mathsf{F}\ ((id \times \mathsf{s}) \cdot \mathsf{a}) \cdot (\tau_r \cdot (\ulcorner f \urcorner \times id) \cdot \mathsf{xr}) \bullet (\tau_r \cdot (p \times id) \cdot \langle id, \pi_2\rangle) \\
= & \quad \{\ \ulcorner\cdot\urcorner\ \text{def};\ \cdot\ /\ \bullet\ \text{assoc}\ \} \\
& \eta \cdot (id \times \pi_1) \cdot (id \times \mathsf{s}) \cdot \mathsf{a} \bullet (\tau_r \cdot (\ulcorner f \urcorner \times id) \cdot \mathsf{xr}) \bullet (\tau_r \cdot (p \times id) \cdot \langle id, \pi_2\rangle) \\
= & \quad \{\ (id \times \pi_1) \cdot (id \times \mathsf{s}) \cdot \mathsf{a} = \pi_1 \cdot \mathsf{xr};\ \cdot\ /\ \bullet\ \text{assoc}\ \} \\
& \eta \cdot \pi_1 \cdot \bullet \mathsf{F}\ \mathsf{xr} \cdot (\tau_r \cdot (\ulcorner f \urcorner \times id) \cdot \mathsf{xr}) \bullet (\tau_r \cdot (p \times id) \cdot \langle id, \pi_2\rangle) \\
= & \quad \{\ extr\ \text{def}\ \} \\
& \eta \cdot \pi_1 \cdot \bullet extr\ \ulcorner f \urcorner \bullet (\tau_r \cdot (p \times id) \cdot \langle id, \pi_2\rangle) \\
= & \quad \{\ \eta \cdot \pi_1 \bullet extr\ \ulcorner f \urcorner = \eta \cdot \pi_1\ \} \\
& \eta \cdot \pi_1 \bullet (\tau_r \cdot (p \times id) \cdot \langle id, \pi_2\rangle) \\
= & \quad \{\ \cdot\ /\ \bullet\ \text{assoc};\ \bullet\ \text{identity}\ \} \\
& \mathsf{F}\ \pi_1 \cdot \tau_r \cdot (p \times id) \cdot \langle id, \pi_2\rangle \\
= & \quad \{\ \mathsf{F}\ \pi_1 \cdot \tau_r = \pi_1\ \} \\
& \pi_1 \cdot (p \times id) \cdot \langle id, \pi_2\rangle \\
= & \quad \{\ \pi_1\ \text{natural};\ \times\ \text{cancel}\ \} \\
& p \\
& \square
\end{aligned}$$

B `oop2mmm` Example

As a proof-of-concept for the semantics explained in this report, we wrote a Haskell tool that translates the same toy object oriented language into monadic Mealy Machines.

Although the tool is available for the community, it is still highly experimental, and we provide no guarantees whatsoever. In this section we'll explain the usage and provide some examples.
In general, the tool is fairly simple to use. Let's say that we have the contents of listing 1.1 into a file called *obj_stack.mmm* (this file can be found in the Examples folder, if you download the source).

Listing 1.1. Stack Component

```
class Stack {
  var st : list of int;

  push(i : int) : void {
    this.st = i @ this.st;
  }

  top() : int {
    res = 0;
    if (length this.st == 0)
      then { res = 0; }
      else { res = head(this.st); }
    return res;
  }

  pop() : void {
    if (length this.st > 0)
      then { this.st = tail(this.st); }
  }
}
```

Listing 1.2. Simplified Stack Code

```
var st : list of int;

push (i : int) : void {
  GET(this.st, _pure4);
  _liftset9 = ((:) i _pure4);
  SET(this.st, _liftset9);
  return NIL;
}
top () : int {
  GET(this.st, _pure5);
  _explin2 = (length _pure5);
  _parmlin0 = (_explin2 == 0);
  res = 0;
  if _parmlin0
  then {
    res = 0;
  }
  else {
    GET(this.st, _pure6);
    res = (head _pure6);
    COLLECT(_pure6)
  }
  return res;
}
pop () : void {
  GET(this.st, _pure7);
  _explin3 = (length _pure7);
  _parmlin1 = (_explin3 > 0);
  if _parmlin1
  then {
     GET(this.st, _pure8);
     _liftset10 = (tail _pure8);
     SET(this.st, _liftset10);
     COLLECT(_liftset10, _pure8)
   }
  return NIL;
}
```

Running the command

```
$ oop2mmm --output="Stack.hs" --dump MMM/Examples/obj_stack.mmm
```

Will dump the simplified code (listing 1.2) and write the Haskell equivalent (part of the code is shown in figure 4) of the Stack component to *Stack.hs*.
We can actually run some snipets and see that the code is exetucing correctly:

```
*Stack> stackPUSH ([2,3,4], 1)
([1,2,3,4],())
*Stack> stackTOP ([1,2,3,4], ())
([1,2,3,4],1)
*Stack> stackPOP ([1,2,3,4], ())
([2,3,4],())
*Stack> stackPOP ([], ())
([],())
*Stack> stackTOP ([], ())
([],0)
```

Fig. 4. (Part of) Haskell-encoded MMM of the Stack Component

```
module Stack where

import MMM.Core.All

{-# LINE 2 "MMM/Examples/obj_stack.mmm" #-}
set_st_Stack :: (Strong m) => (MMM m [Integer] [Integer] ())
set_st_Stack = (return . (split p2 bang))

{-# LINE 2 "MMM/Examples/obj_stack.mmm" #-}
get_st_Stack :: (Strong m) => (MMM m [Integer] () [Integer])
get_st_Stack = ((f2m id) .! getst)

{-# LINE 4 "MMM/Examples/obj_stack.mmm" #-}
stackPUSH :: (Strong m) => (MMM m [Integer] Integer ())
stackPUSH = (f2m (p1 . _a3))
            .! (runm_ set_st_Stack)
               .! (f2m _a2)
            .! (runm (f2m (\ (i , _pure4) -> ((:) i _pure4))))
               .! (f2m _a1)
            .! (runm get_st_Stack)
               .! (f2m _a0)
            .! (f2m nilpw)
  where
    -- (NIL, i)
    --   -> (NIL, i)
    _a0 = (split p1 p2)

    -- (NIL, (i, _pure4))
    --   -> ((i, _pure4), NIL)
    _a1 = (split (split (p1 . p2) (p2 . p2)) (id . p1))

    -- ((i, _pure4), (NIL, _liftset9))
    --   -> (_liftset9, ((i, _pure4), NIL))
    _a2 = (split (p2 . p2) (split (split (p1 . p1) (p2 . p1)) (p1 . p2)))

    -- (_liftset9, ((i, _pure4), NIL))
    --   -> (NIL, ((_liftset9, i), _pure4))
    _a3 = (split (p2 . p2) (split (split (id . p1) ((p1 . p1) . p2)) ((p2 . p1) . p2)))


stackTOP :: (Strong m) => (MMM m [Integer] () Integer)
stackTOP = (f2m (p1 . _a8))
           .! (mcond
               ((p2 . p1) . p2)
               ((f2m ((split (split (split (split (id . p1) (((p1 . p1) . p1) . p2)) ((p2 . (p1 . p1)) . p2)) ((p2 . p1)!
! . p2)) (p2 . p2)) . ((p2 >< id) . (split (split ((p2 . p1) . p2) (p2 . p2)) (split (split (split (id . p1) ((((p1 . p1)!
! . p1) . p1) . p2)) ((p2 . ((p1 . p1) . p1)) . p2)) ((p2 . (p1 . p1)) . p2))))))
                .! (runm (f2m (const 0)))
                .! (f2m _a4))
               ((f2m (split (split (split (split (((p1 . p1) . p1) . p1) (p2 . ((p1 . p1) . p1))) (p2 . (p1 . p1))) (p2 !
!. p1)) p2))
                .! (f2m _c7)
                   .! (f2m ((split (split (split (split (split (id . p1) ((((p1 . p1) . p1) . p1) . p2)) ((p2 . ((p1 . p!
!1) . p1)) . p2)) ((p2 . (p1 . p1)) . p2)) ((p2 . p1) . p2)) (p2 . p2)) . ((p2 >< id) . (split (split ((p2 . p1) . p2) (p!
!2 . p2)) (split (split (split (split (id . p1) (((((p1 . p1) . p1) . p1) . p2)) ((p2 . (((p1 . p1) . p1) . p1)) . !
!p2)) ((p2 . ((p1 . p1) . p1)) . p2)) ((p2 . (p1 . p1)) . p2))))))
                      .! (runm (f2m (\ _pure6 -> (head _pure6))))
                      .! (f2m _a6)
                   .! (runm get_st_Stack)
                      .! (f2m _a5)))
           .! (runm (f2m (const 0)))
              .! (f2m _a3)
           .! (runm (f2m (\ _explin2 -> (_explin2 == 0))))
              .! (f2m _a2)
           .! (runm (f2m (\ _pure5 -> (length _pure5))))
              .! (f2m _a1)
           .! (runm1 get_st_Stack)
              .! (f2m _a0)
           .! (f2m id)
  where
    -- NIL
    --   -> (NIL,)
    _a0 = id

    -- (NIL, _pure5)
    --   -> (_pure5, NIL)
    _a1 = (split (id . p2) (id . p1))

    -- (_pure5, (NIL, _explin2))
    --   -> (_explin2, (_pure5, NIL))
    _a2 = (split (p2 . p2) (split (id . p1) (p1 . p2)))

    -- (_explin2, ((_pure5, NIL), _parmlin0))
    --   -> (NIL, ((_explin2, _pure5), _parmlin0))
    _a3 = (split ((p2 . p1) . p2) (split (split (id . p1) ((p1 . p1) . p2)) (p2 . p2)))

    -- (NIL, (((_explin2, _pure5), _parmlin0), res))
    --   -> (NIL, (((_explin2, _pure5), _parmlin0), res))
    _a4 = (split (id . p1) (split (split (split (((p1 . p1) . p1) . p2) ((p2 . (p1 . p1)) . p2)) ((p2 . p1) . p2)) (p2 .!
! p2)))

    -- (NIL, (((_explin2, _pure5), _parmlin0), res))
    --   -> (NIL, (((_explin2, _pure5), _parmlin0), res))
    _a5 = (split (id . p1) (split (split (split (((p1 . p1) . p1) . p2) ((p2 . (p1 . p1)) . p2)) ((p2 . p1) . p2)) (p2 .!
! p2)))

    -- (NIL, ((((_explin2, _pure5), _parmlin0), res), _pure6))
    --   -> (_pure6, ((((NIL, _explin2), _pure5), _parmlin0), res))
    _a6 = (split (p2 . p2) (split (split (split (split (id . p1) ((((p1 . p1) . p1) . p1) . p2)) ((p2 . ((p1 . p1) . p1)!
!) . p2)) ((p2 . (p1 . p1)) . p2)) ((p2 . p1) . p2)))

    -- (((((res, _pure6), NIL), _explin2), _pure5), _parmlin0)
    --   -> ((((res, NIL), _explin2), _pure5), _parmlin0)
    _c7 = (p2 . (split (p2 . (((p1 . p1) . p1) . p1)) (split (split (split (split ((((p1 . p1) . p1) . p1) . p1) (p2 . (!
!(p1 . p1) . p1))) (p2 . (p1 . p1))) (p2 . p1)) p2)))

    -- ((((res, NIL), _explin2), _pure5), _parmlin0)
    --   -> (res, (((NIL, _explin2), _pure5), _parmlin0))
    _a8 = (split (((p1 . p1) . p1) . p1) (split (split (split (p2 . ((p1 . p1) . p1)) (p2 . (p1 . p1))) (p2 . p1)) p2))
```