# *The Algebra of Programming in Haskell*

Bruno Oliveira

# Datatype Generic Programming - Motivation/Goals

- The project is to develop a novel mechanism for parameterizing programs, namely parametrization by a datatype or type constructor.
- We aim to develop a calculus for constructing datatype-generic programs.
- Ultimate goal of improving the state of the art in generic object-oriented programming, as occurs for example in the C++ Standard Template Library.

# Introduction - Algebra of Programming

In the excellent book *Algebra of Programming*, Bird and de Moor show us how to *calculate programs* in a very elegant way. Further, the problems that they solve are datatype-generic. As they note:

> *"... The problems are abstract in the sense that they are parameterized by one or more datatypes. ..."*

The Algebra of Programming provides us:

- A mathematical framework based in a *categorical calculus of relations*
- The categorical calculus allow us to formulate algorithmic strategies without reference to specific datatypes.
- An important subset of generic functions.

# Notation

| | |
|---|---|
| $f \circ g$ | function composition |
| $id$ | identity function |
| $\underline{k}$ | constant function |
| $f^{\rightarrow}$ | curry function |
| $f^{\times}$ | uncurry function |
| $i_1$ | left injection to sum |
| $i_2$ | right injection to sum |
| $\pi_1$ | left component of product |
| $\pi_2$ | right component of product |
| $1$ | unit type and value |
| $f \vartriangle g$ | fork over product |
| $f \triangledown g$ | either function |
| $f + g$ | sum mapping |
| $f \times g$ | product mapping |

# A Theory of Lists

Consider the Haskell $[A]$ (we use capitals instead of lower case to denote types) datatype. A possible definition for it, could be:

$$\mathbf{data}\ [A] = [\,]\ |\ A : [A]$$

You can view this data definition as the following isomorphism:

$$[A] \underset{in_{[\,]}}{\overset{out_{[\,]}}{\underset{\cong}{\rightleftarrows}}} 1 + A \times [A]$$

# A Theory of lists

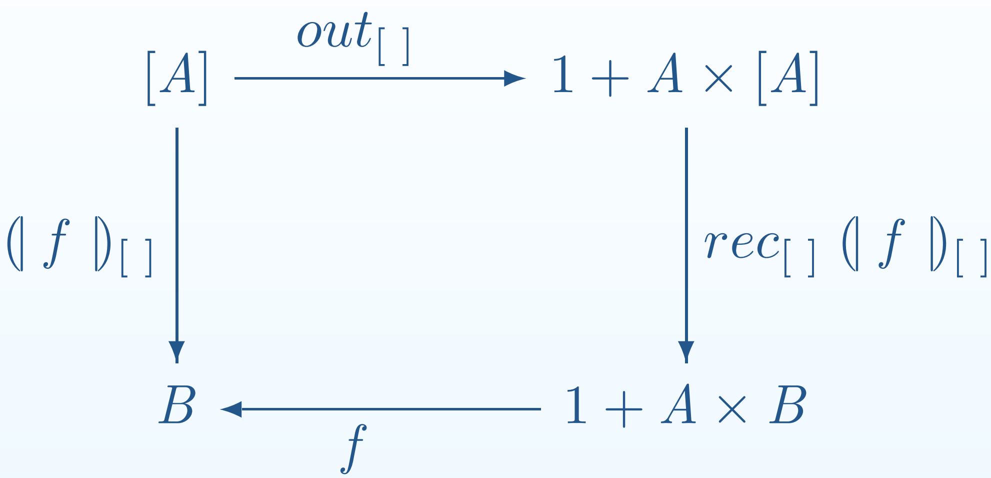
A well known function on lists is the $foldr$ function:

$$foldr\ f\ k\ [\,] = k$$
$$foldr\ f\ k\ (x : xs) = f\ x\ (foldr\ f\ k\ xs)$$

$$[A] \xrightarrow{foldr\ f\ k} B$$

$foldr$ and its dual $unfoldr$ are the basis for many definitions on lists. "Uncurried" versions of this functions, are the basis for much of the theory presented in the book.

# A Theory of Lists - Morphisms




We call *catamorphism* to the "uncurried" version of $foldr$ and we denote it as $(\!| f |\!)_{[\,]}$.

$$(\!| f |\!)_{[\,]} = f \circ rec_{[\,]}\ (\!| f |\!)_{[\,]} \circ out_{[\,]}$$

**where**

$$out_{[\,]} = (\underline{1} + head \vartriangle tail) \circ (\equiv [\,])?$$

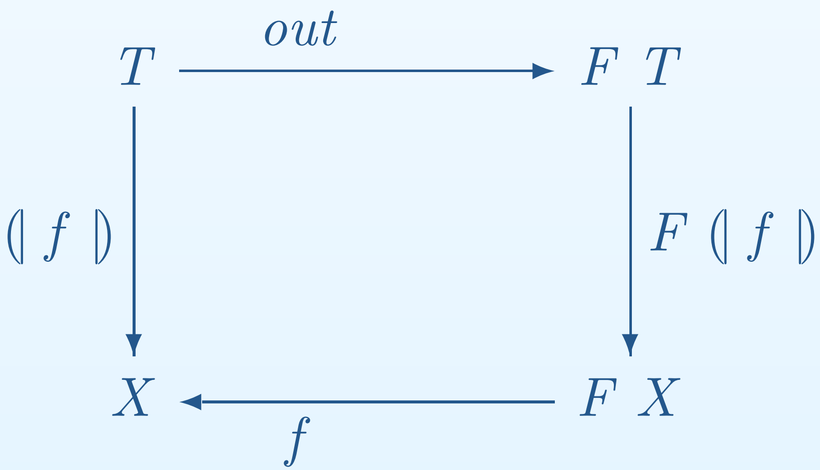
$$rec_{[\,]}\ g = id + id \times g$$

# Functors - Generalizing the Theory

By using *functors*, we can generalize the theory. For instance, we could abstract the *expansion* of $[A]$ to:

$$F\ X \ \cong 1 + A \times X$$

Parameterizing $F$ with $[A]$, we would obtain $1 + A \times [A]$. A catamorphism could be expressed generically by:



# Functional Dependencies

Allow programmers to specify multiple parameter classes more precisely. For instance:

$$\textbf{class}\ C\ a\ b$$
$$\textbf{class}\ D\ a\ b \mid a \to b$$
$$\textbf{class}\ E\ a\ b \mid a \to b, b \to a$$

From these definitions we can tell that:

- Class $C$ is a binary relation.
- Class $D$ is not only a relation, but actually a (partial) function.
- Class $E$ represents a (partial) one-one mapping.

# Related Work - PolyP

## PolyP

The original PolyP system allows us to write generic definitions for regular datatypes of kind $* \rightarrow *$. The system works by using a type based translation from PolyP to Haskell at compile time.

## PolyP 2

More recently, PolyP 2 introduces a novel translation mechanism allowing PolyP code to be translated to Haskell classes and instances. The structure of a regular datatype is described by its *pattern functor*. For instance:

$$\mathbf{data}\ List\ a = Nil \mid Cons\ a\ (List\ a)$$
$$\mathbf{type}\ ListF = Empty + Par \times Rec$$

# Related Work - PolyP 2

All pattern functors (except $\rightarrow$) are instances of the class $P\_fmap2$:

$$\textbf{class } P\_fmap2\ f \textbf{ where}$$
$$fmap2 :: (a \rightarrow c) \rightarrow (b \rightarrow d) \rightarrow (f\ a\ b \rightarrow f\ c\ d)$$

To convert between a datatype and its pattern functor, the multi-parameter type class $FunctorOf$ is used:

$$\textbf{class } FunctorOf\ f\ d \mid d \rightarrow f \textbf{ where}$$
$$inn :: f\ a\ (d\ a) \rightarrow d\ a$$
$$out :: d\ a \rightarrow f\ a\ (d\ a)$$

Having these, we could define, for instance:

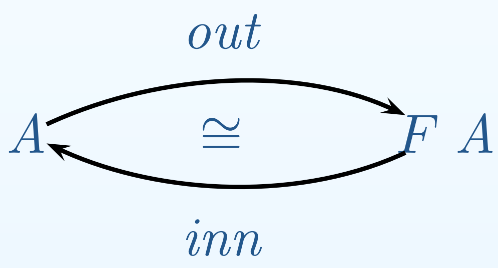
$$(\!|\ f\ |\!) = f \circ fmap2\ id\ (\!|\ f\ |\!) \circ out$$
$$[\!(\ f\ )\!] = inn \circ fmap2\ id\ [\!(\ f\ )\!] \circ f$$

# APLib

In the Algebra of Programming Library (APLib) we show a similar framework working for regular datatypes of all kinds.

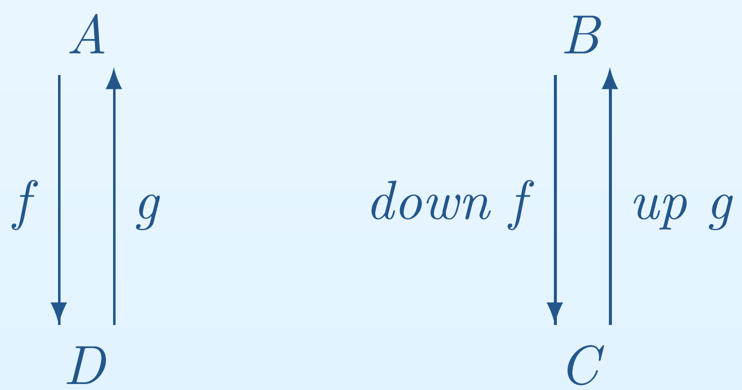
The class $Iso$ acts like an "weak" *isomorphism* by establishing an one-one mapping between $A$ and $B$

$$\textbf{class } Iso\ a\ b \mid a \rightarrow b, b \rightarrow a \textbf{ where}$$
$$out :: a \rightarrow b$$
$$inn :: b \rightarrow a$$




Class $MorphArrows$ contains more information than $Functor$.

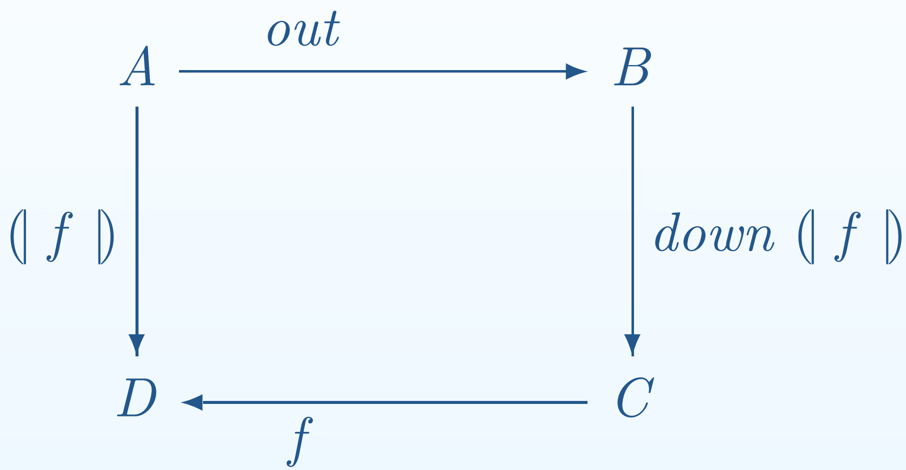
$$\textbf{class } Iso\ a\ b \Rightarrow MorphArrows\ a\ b\ c\ d$$
$$\mid a\ b\ d \rightarrow c, a\ b\ c \rightarrow d \textbf{ where}$$
$$down :: (a \rightarrow d) \rightarrow b \rightarrow c$$
$$up :: (d \rightarrow a) \rightarrow c \rightarrow b$$



# APLib - Morphisms

By using $MorphArrows$ we can define *catamorphisms* as:

$$(\!|\ f\ |\!) = f \circ down\ (\!|\ f\ |\!) \circ out$$




Defining *anamorphisms* and *hylomorphisms* is easy:

$$[\!(\ f\ )\!] = inn \circ up\ [\!(\ f\ )\!] \circ f$$

$$[\![\ f, g\ ]\!] = (\!|\ f\ |\!) \circ [\!(\ g\ )\!]$$

# APLib - Example

$$\textbf{data}\ \mathit{Expr}\ \mathit{op}\ a = \mathit{Leaf}\ a$$
$$\quad \mid \mathit{Binary}\ \mathit{op}\ (\mathit{Expr}\ \mathit{op}\ a)\ (\mathit{Expr}\ \mathit{op}\ a)$$

$$\textbf{data}\ \mathit{Op} = \mathit{Sum} \mid \mathit{Sub}$$

$$\textbf{instance}\ \mathit{MorphArrows}$$
$$\quad (\mathit{Expr}\ \mathit{op}\ a)$$
$$\quad (a + \mathit{op} \times \mathit{Expr}\ \mathit{op}\ a \times \mathit{Expr}\ \mathit{op}\ a)$$
$$\quad (a + \mathit{op} \times b \times b)$$
$$\quad b\ \textbf{where}$$
$$\quad \mathit{down}\ f = \mathit{id} + \mathit{id} \times f \times f$$
$$\quad \mathit{up}\ f = \mathit{id} + \mathit{id} \times f \times f$$

Calculating the value of an expression:

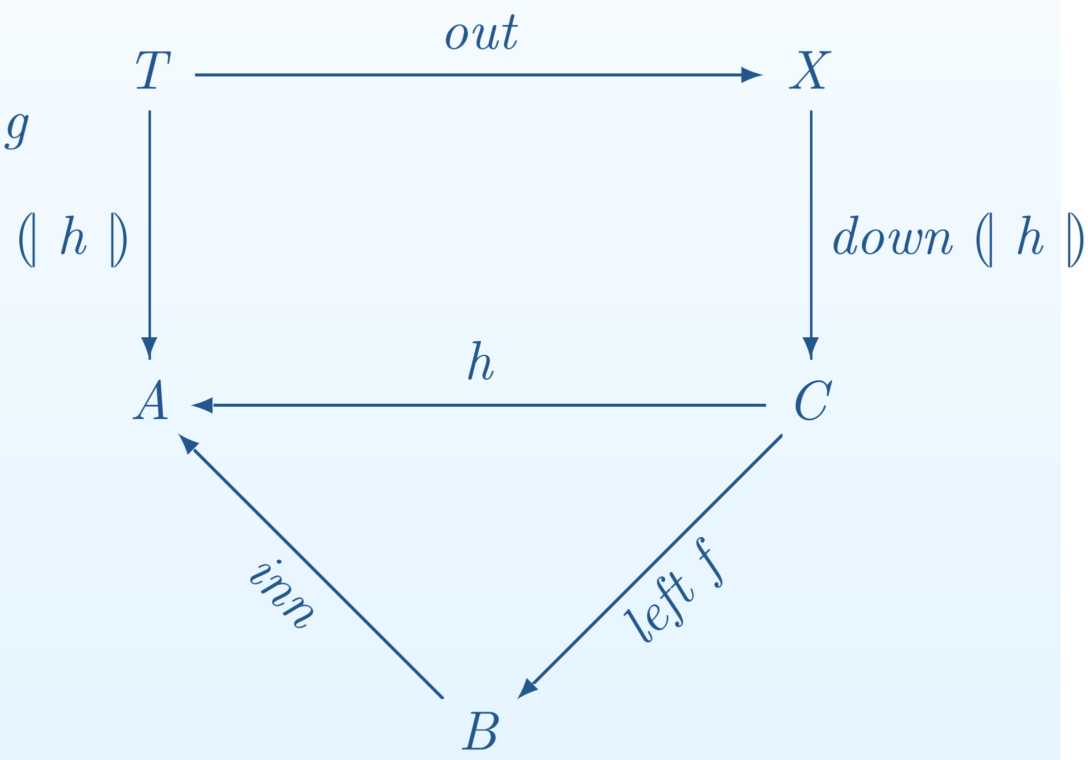
$$\mathit{eval} :: \mathit{Expr}\ \mathit{Op}\ \mathit{Int} \rightarrow \mathit{Int}$$
$$\mathit{eval} = (\!|\ \mathit{id} \triangledown \mathit{evalOp}\ |\!)$$
$$\quad \textbf{where}$$
$$\qquad \mathit{evalOp} = ((+)^{\times} \circ \pi_2 \triangledown (-)^{\times} \circ \pi_2) \circ (\mathit{isSum} \circ \pi_1)?$$

# APLib - Defining Generic Map

We can define a generic map by having a class $MapArrows$ which transforms the $Functor$ that we are working with. The parameters $f$ and $g$ are similar to *kind-indexed types*.

$$\textbf{class}\ Iso\ a\ b \Rightarrow MapArrows\ a\ b\ c\ f\ g$$
$$\quad \mid a\ b\ c \rightarrow f\ g\ \textbf{where}$$
$$\quad left :: f \rightarrow c \rightarrow b$$
$$\quad right :: g \rightarrow b \rightarrow c$$

$$gmap\ f = (\!|\ h\ |\!)$$
$$\quad \textbf{where}$$
$$\qquad h = inn \circ left\ f$$



# Specializations

Two possible approaches to specializations:

1. By Type - define a new function with a more restrictive type. Usefull for having less generic functions.

$$cata_{*\to *} :: MorphArrows\ (f\ \ a)\ u\ c\ b \Rightarrow (c \to b) \to f\ \ a \to b$$
$$cata_{*\to *}\ g = (\!|\ g\ |\!)$$

2. By Definition - define a new function based on the definition of the most generic one, but specific to a type. Useful for optimization.

$$out_{[\,]} = (\underline{1} + head \vartriangle tail) \circ (\equiv [\,])?$$
$$down_{[\,]}\ g = id + id \times g$$
$$(\!|\ f\ |\!)_{[\,]} = f \circ down_{[\,]}\ (\!|\ f\ |\!)_{[\,]} \circ out_{[\,]}$$

# Abstract Data Types

$$\textbf{data}\ Ord\ a \Rightarrow BTree\ a =$$
$$\quad Empty$$
$$\quad \mid Branch\ a\ (BTree\ a)\ (BTree\ a)$$

$$\begin{aligned}
&\textbf{class}\ OrdList\ f\ \textbf{where}\\
&\quad isNil :: f\ a \rightarrow Bool\\
&\quad nil :: f\ a\\
&\quad add :: Ord\ a \Rightarrow a \rightarrow f\ a \rightarrow f\ a\\
&\quad getNext :: Ord\ a \Rightarrow f\ a \rightarrow Maybe\ (a, f\ a)
\end{aligned}$$

The instance for $BTree\ a$ could be:

$$\begin{aligned}
&\textbf{instance}\ (OrdList\ f, Ord\ a) \Rightarrow Iso\ (f\ a)\ (1 + a \times f\ a)\ \textbf{where}\\
&\quad out = (\underline{1} + fromJust \circ getNext) \circ isNil?\\
&\quad inn = (\underline{nil} \bigtriangledown add^{\times})
\end{aligned}$$

Given that, we could define a sorting function:

$$\begin{aligned}
&sort :: [Int] \rightarrow [Int]\\
&sort = (\!|\ inn\ |\!) \circ ([\!(\ out\ )\!] :: [Int] \rightarrow BTree\ Int)
\end{aligned}$$

# Future Research

- Generate the instance for $MorphArrows$ and $MapArrows$ automatically. *Template Haskell* seems to fit well. A mechanism like *Derivable type classes* might be another possibility.
- Try to minimize the number of classes/instances.
- Consider a larger range of datatypes: Ian Bailey and Paul Blampied work.
- Consider using the framework in a dependent type system.

# Future and Related Work - Type Transformers

Type transformers allow us define types and definitions based on types. For instance, for $out$ we could have:

The type is given by:

$$\theta < Type > :: Type$$

The definition is given by:

$$out < T > :: T \rightarrow \theta < T >$$

This would fit nicely into classes with functional dependencies.

$$\begin{array}{l} \textbf{class } Iso\ T\ \theta \mid T \rightarrow\ \theta,\ \theta \rightarrow T \textbf{ where} \\ \quad out < T > :: T \rightarrow\ \theta \\ \quad inn <\ \theta, T > :: \theta \rightarrow T \end{array}$$

When defining $out$, we will be interested in matching the recursive pattern of $T$ in $F\ T$.

$$out < T > :: T \rightarrow \theta$$
$$out < Data\ \ T > = out' < T, Data\ \ T >$$

# Future and related work - Type Transformers

$$out' < T, Rec > :: T \rightarrow \ \theta$$
$$out' < Rec, Rec> = id$$
$$out' < 1, \_> = \underline{()}$$
$$out' < Prim, \_> = id$$
$$out' < Data\ \ t, Rec> = t\ \ out' < t, Rec>$$
$$out' < a + b, Rec> = out' < a, Rec > + out' < a, Rec>$$
$$out' < a \times b, Rec> = out' < a, Rec > \times out' < b, Rec>$$
$$out' < Con\ \ c, Rec> = out' < a, Rec > \circ isC?$$

$$\theta < T, Rec > :: Type$$
$$\theta < Rec, Rec> = Rec$$
$$\theta < 1, \_> = ()$$
$$\theta < Prim, \_> = Prim$$
$$\theta < Data\ \ t, Rec> = t\ \theta < t, Rec>$$
$$\theta < a + b, Rec> = \theta < a, Rec > + \theta < a, Rec>$$
$$\theta < a \times b, Rec> = \theta < a, Rec > \times \theta < b, Rec>$$
$$\theta < Con\ \ c\ \ a, Rec> = \theta < a, Rec > \circ isC?$$

# Conclusions

- Theory based on categorical calculus of relations allows us to reason about the programs.
- Integrates nicely with other features of Haskell (ex. type classes)
- Possible application for optimization.
- Support for regular datatypes with no restriction on the kind.
- Restricted support for generic functions.
- Still not "quite" right: no explicit Functor concept, need for dual definitions.