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.
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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f \circ g$</td>
<td>function composition</td>
</tr>
<tr>
<td>$id$</td>
<td>identity function</td>
</tr>
<tr>
<td>$k$</td>
<td>constant function</td>
</tr>
<tr>
<td>$f \rightarrow$</td>
<td>curry function</td>
</tr>
<tr>
<td>$f \times$</td>
<td>uncurry function</td>
</tr>
<tr>
<td>$i_1$</td>
<td>left injection to sum</td>
</tr>
<tr>
<td>$i_2$</td>
<td>right injection to sum</td>
</tr>
<tr>
<td>$\pi_1$</td>
<td>left component of product</td>
</tr>
<tr>
<td>$\pi_2$</td>
<td>right component of product</td>
</tr>
<tr>
<td>$1$</td>
<td>unit type and value</td>
</tr>
<tr>
<td>$f \triangle g$</td>
<td>fork over product</td>
</tr>
<tr>
<td>$f \triangledown g$</td>
<td>either function</td>
</tr>
<tr>
<td>$f + g$</td>
<td>sum mapping</td>
</tr>
<tr>
<td>$f \times g$</td>
<td>product mapping</td>
</tr>
</tbody>
</table>
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:

$$\text{data } [A] = [ ] | A : [A]$$

You can view this data definition as the following isomorphism:

$$[A] \cong 1 + A \times [A]$$
A Theory of lists

A well known function on lists is the \( \text{foldr} \) function:

\[
\begin{align*}
\text{foldr} f k [] &= k \\
\text{foldr} f k (x : xs) &= f x (\text{foldr} f k xs)
\end{align*}
\]

\( \text{foldr} \) and its dual \( \text{unfoldr} \) 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 \textit{catamorphism} to the "uncurried" version of \textit{foldr} and we denote it as \( ( f )_[] \).

\[
( f )_[] = f \circ rec_[] \circ ( f )_[] \circ out_[
\]

where

\[
out_[] = (1 + \text{head} \triangle \text{tail}) \circ (\equiv [])?
\]

\[
rec_[] g = id + id \times g
\]
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:

\[
\begin{align*}
T & \xrightarrow{\text{out}} F T \\
\downarrow & \quad & \downarrow \\
\langle f \rangle & \quad & F \langle f \rangle \\
\downarrow & \quad & F X \\
X & \xleftarrow{f} & F X
\end{align*}
\]
Functional Dependencies

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

```haskell
class C a b
class D a b | a → b
class E a b | a → b, b → 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:

```haskell
data List a = Nil | Cons a (List a)
type ListF = Empty + Par \times Rec
```
Related Work - PolyP 2

All pattern functors (except →) are instances of the class \( P_{fmap2} \):

\[
\text{class } P_{fmap2} \ f \ \text{where} \\
\quad \text{fmap2} :: (a \rightarrow c) \rightarrow (b \rightarrow d) \rightarrow (f \ a \ b \rightarrow f \ c \ d)
\]

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

\[
\text{class } FunctorOf \ f \ d \ | \ d \rightarrow f \ \text{where} \\
\quad \text{inn} :: f \ a \ (d \ a) \rightarrow d \ a \\
\quad \text{out} :: d \ a \rightarrow f \ a \ (d \ a)
\]

Having these, we could define, for instance:

\[
(\Downarrow f \Uparrow) = f \circ \text{fmap2} \ id \ (\Downarrow f \Uparrow) \circ \text{out} \\
[ f ] = \text{inn} \circ \text{fmap2} \ id \ [ f ] \circ f
\]
In the Algebra of Programming Library (APLib) we show a similar framework working for regular datatypes of all kinds.

The class \textit{Iso} acts like an "weak" isomorphism by establishing an one-one mapping between \( A \) and \( B \).

\[
\text{class Iso } a \; b \mid a \rightarrow b, \; b \rightarrow a \ \text{where}
\]
\[
\begin{align*}
\text{out} & :: a \rightarrow b \\
\text{inn} & :: b \rightarrow a
\end{align*}
\]

\textbf{Class} \textit{MorphArrows} \textbf{contains more information than Functor}.

\[
\text{class Iso } a \; b \Rightarrow \textit{MorphArrows} \ a \; b \; c \; d \\
\mid a \; b \; d \rightarrow c, \ a \; b \; c \rightarrow d \ \text{where}
\]
\[
\begin{align*}
\text{down} & :: \ (a \rightarrow d) \rightarrow b \rightarrow c \\
\text{up} & :: \ (d \rightarrow a) \rightarrow c \rightarrow b
\end{align*}
\]
By using \textit{MorphArrows} we can define \textit{catamorphisms} as:

\[
(j \circ f \circ j) = f \circ \text{down}(j \circ f \circ j) \circ \text{out}
\]

Defining \textit{anamorphisms} and \textit{hylomorphisms} is easy:

\[
\llbracket f \rrbracket = \text{inn} \circ \text{up} \circ \llbracket f \rrbracket \circ f
\]

\[
\llbracket f, g \rrbracket = (\llbracket f \rrbracket) \circ \llbracket g \rrbracket
\]
data Expr op a = Leaf a | Binary op (Expr op a) (Expr op a)

data Op = Sum | Sub

Calculating the value of an expression:

\[
eval :: Expr \text{ Op } Int \rightarrow Int
\]
\[
eval = (\mid id \downarrow evalOp \mid)
\]
where
\[
evalOp = ((+)^{\times} \circ \pi_2 \downarrow (-)^{\times} \circ \pi_2) \circ (isSum \circ \pi_1)\
\]
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*.

```haskell
class Iso a b ⇒ MapArrows a b c f g
  | a b c → f g where
  left :: f → c → b
  right :: g → b → c

gmap f = (\ h \)
  where
  h = inn ∘ left f
```

![Diagram showing the transformation of types](https://via.placeholder.com/150)
Specializations

Two possible approaches to specializations:

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

   \[ \text{cata} : \text{MorphArrows} (f \ a) u c b \Rightarrow (c \rightarrow b) \rightarrow f \ a \rightarrow b \]
   \[ \text{cata} g = (\| g \|) \]

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

   \[ \text{out} [\ ] = (1 + \text{head } \triangle \text{tail}) \circ (\equiv [\ ]) ? \]
   \[ \text{down} [\ ] g = \text{id} + \text{id} \times g \]
   \[ (\| f \|) [\ ] = f \circ \text{down} [\ ] (\| f \|) [\ ] \circ \text{out} [\ ] \]
Abstract Data Types

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

\textbf{class} \quad \textit{OrdList f} \quad \textbf{where}

\textit{isNil} :: f a \rightarrow \textit{Bool}
\textit{nil} :: f a
\textit{add} :: \textit{Ord a} \Rightarrow a \rightarrow f a \rightarrow f a
\textit{getNext} :: \textit{Ord a} \Rightarrow f a \rightarrow \textit{Maybe} (a, f a)

The instance for \textit{BTree a} could be:

\textbf{instance} \quad (\textit{OrdList f}, \textit{Ord a}) \Rightarrow \textit{Iso (f a) (1 + a \times f a)} \quad \textbf{where}

\textit{out} = (1 + \text{fromJust} \circ \text{getNext}) \circ \text{isNil}?
\textit{inn} = (\text{nil} \triangledown \text{add}^\times)

Given that, we could define a sorting function:

\textit{sort} :: [\textit{Int}] \rightarrow [\textit{Int}]
\textit{sort} = (\mid \text{inn} \mid) \circ ([\text{out}] :: [\textit{Int}] \rightarrow \textit{BTree} \textit{Int})
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.
Type transformers allow us define types and definitions based on types. For instance, for \texttt{out} we could have:

The type is given by:

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

The definition is given by:

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

This would fit nicely into classes with functional dependencies.

```haskell
class Iso T \theta \mid T \to \theta, \theta \to T \ where
\quad out \ < \ T \ > :: T \to \theta
\quad inn \ < \ \theta, T \ > :: \theta \to T
```

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

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

$$out \ < \ Data \ T \ > = out' \ < \ T, Data \ T \ >$$
Future and related work - Type Transformers

\[
\begin{align*}
out' & < T, \text{Rec} > :: T \rightarrow \theta \\
out' & < \text{Rec}, \text{Rec} > = id \\
out' & < 1, \_ > = () \\
out' & < \text{Prim}, \_ > = id \\
out' & < \text{Data} \ t, \text{Rec} > = t \ out' < t, \text{Rec} > \\
out' & < a + b, \text{Rec} > = out' < a, \text{Rec} > + out' < a, \text{Rec} > \\
out' & < a \times b, \text{Rec} > = out' < a, \text{Rec} > \times out' < b, \text{Rec} > \\
out' & < \text{Con c} a, \text{Rec} > = out' < a, \text{Rec} > \circ isC
\end{align*}
\]

\[
\begin{align*}
\theta & < T, \text{Rec} > :: \text{Type} \\
\theta & < \text{Rec}, \text{Rec} > = \text{Rec} \\
\theta & < 1, \_ > = () \\
\theta & < \text{Prim}, \_ > = \text{Prim} \\
\theta & < \text{Data} \ t, \text{Rec} > = t \ \theta < t, \text{Rec} > \\
\theta & < a + b, \text{Rec} > = \theta < a, \text{Rec} > + \theta < a, \text{Rec} > \\
\theta & < a \times b, \text{Rec} > = \theta < a, \text{Rec} > \times \theta < b, \text{Rec} > \\
\theta & < \text{Con c} a, \text{Rec} > = \theta < a, \text{Rec} > \circ isC
\end{align*}
\]
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.