Generic Container Programming in Haskell

Bruno Oliveira

Introduction

One of the goals for this first year is:

- capture existing template meta-programming techniques within datatype-generic programming

The main goal of this work, is to check for possible features in programming languages for doing generic programming.

For doing that, we'll try to explore STL Iterators and Concepts. Moreover, we will, more generically, explore some morphisms over containers.

Generic Programming in Haskell

Currently, there are a few projects in the functional programming community that are trying to do (or are related somehow) to generic programming. We point out some of them:

- Scrap your boilerplate
- Template Haskell
- Generic Haskell
- Dependent Types

Generic Programming in Haskell

Technique	Interpreter		Compiler	
	GF	DTT	GF	DTT
Scrap your boilerplate	Limited*	No	Limited*	No
Template Haskell	Yes**	Yes**	Spec	Spec
Generic Haskell	Yes	No	Spec	No
Dependent Types	Yes	Yes	Yes	Yes

* Generic traversal mechanisms only.

** We assume that Template Haskell can simulate Generic Programming by making use of their template mechanisms.

Scrap your boilerplate

- Scrap boilerplate is an approach that focus on term traversal as the prime idiom of generic programming.
- It aims at a smooth integration of generic programming with Haskell.
- Supports a combinator style of generic programming.
- It is pretty much lightweight version of generic programming when compared to other alternatives (such as Generic Haskell).

The intriguing *gfoldl* :

```
gfoldl :: forall a c.
          (Data a) => (forall a1 b. (Data a1) =>
           c (a1 -> b) -> a1 -> c b) ->
          (forall g. g -> c g) -> a -> c a
```

Polynomial Data-types

Consider the (annotated) polynomial data-type Γ:

$$\Gamma a_0\, a_1 \ldots a_{n-1} = \Pi_0 + \Pi_1 + \ldots + \Pi_{m-1} = \Sigma_0^{m-1}\Pi_i$$

We can split Π_i into Constructor C_i,non-recursive part I_i and recursive part R_i.

$$split\,(\Pi) = C \times I \times R$$

$$map_\Sigma\, split\,(\Sigma_0^{m-1}(\Pi_i)) \; = \Sigma_0^{m-1}(C_i \times I_i \times R_i)$$

Thus, Γ can be represented as:

$$\Gamma\, a_0\, a_1 \ldots \; = C_0 \times I_0 \times R_0 \; + \; \ldots \; = \; \Sigma_0^{m-1} C_i \times I_i \times R_i$$

polynomial Data-types

Consider the generic types Υ, Φ and θ. Υ represents the constructors of the data-type, Φ is the non-recursive information of the equation and θ represents the powers for the recursive part of the equation:

$$\Upsilon = map_{\Sigma}\, fst\, (\Sigma_0^{m-1}(C_i \times I_i \times R_i)) = \Sigma_0^{m-1} C_i$$

$$\Phi = map_{\Sigma}\, snd\, (\Sigma_0^{m-1}(C_i \times I_i \times R_i)) = \Sigma_0^{m-1} I_i$$

$$\theta = map_{\Sigma}\, (powerIndex \circ thr)\, \Sigma_0^{m-1}(C_i \times I_i \times R_i) \;= \Sigma_0^{m-1} Int$$

Polynomial Data-types

We can combine Υ , Φ and θ obtaining:

$$\Omega = map_{\Sigma}\,(id \times id \times powerIndex)\,(\Sigma_0^m(C_i \times I_i \times R_i)) \;= \Sigma_0^m(C_i \times I_i \times Int)$$

Using the first definition of Γ we can apply map-fusion:

$$\Omega = map_{\Sigma}\,((id \times id \times powerIndex) \circ split)\,(\Sigma_0^m(\Pi_i)) \;= \Sigma_0^m(C_i \times I_i \times Int)$$

The operations fst, snd, thr and id are just the same ones we know from Algebra of Programming. The operations powerIndex and split are normal functions applied to a data-type that defines data-types.

```
split :: Prod -> (Constr, Prod, Prod)
powerIndex :: Prod -> Int
powerIndex = length
```

Polynomial container

The generics types Ω, Φ, Υ, θ depend on Γ:

$$\Gamma \rightarrow \Omega, \Gamma \rightarrow \Upsilon, \Gamma \rightarrow \Phi, \Gamma \rightarrow \theta$$

In Haskell it is not possible to express the operations needed, in order to produce those types automatically. However, one can express the dependency relation, thru functional dependencies.
We can express a container as:

```
class Container Γ Φ | Γ → Φ
```

Moreover, this definition allow us to consider a type with kind $(* \rightarrow * \rightarrow ... \rightarrow *)$ as a kind $*$. Φ is, in fact, representing the contents of the container.

Polynomial Container - A few examples

"A container is an Object that stores other Objects (its elements) and that has methods for accessing its elements..."

Def. from "Generic Programming and the STL"

According to what we have before we should have, for some standard data-types, something like:

```
instance Container [a] (1 + a)
instance Container [(a,a)] (1 + a × a)
instance Container (Tree a) (1 + a)
instance Container (ExpTree op a) (a + op + op)
```

However, in the coming sections we will have a more relaxed version:

```
instance Container [a] (a)
instance Container [(a,a)] (a × a)
instance Container (Tree a) (a)
instance Container (ExpTree op a) (a + op)
```

polynomial Data-types - Visitors

Having the two following generic functions:

```
value<Γ,Ω> :: Γ -> Φ
succs<Γ,Ω> :: Γ -> [Γ]
```

- value - retrieves the value of the current element of Γ.
- succs - retrieves the successors of Γ.

One could define a function gvisitor:

```
gvisitor<Γ,Ω> :: (Φ -> c* -> c) -> Γ -> c
```

On this definition `c*`, represents 0 or more `c`'s, the number of `c`'s can be calculated with θ.

polynomial Data-types - Visitors

In Haskell, it is not possible to define generic functions such as value and succs, however Haskell type classes, do allow us to simulate its behavior.

```
class Container Γ Φ => InputIterator Γ Φ where
      value :: Γ -> Maybe Φ
      succs :: Γ -> Maybe [Γ]
```

The definition of gvisitor would be:

```
gvisitor :: InputIterator Γ Φ =>
            (Maybe Φ -> [c] -> c) -> Γ -> c
gvisitor f x =
     case succs x of
          Nothing  -> f (value x) []
          Just s -> f (value x) (map (gvisitor f) s)
```

polynomial Data-types - Visitors

Using the previous definition of gvisitor, one could define a whole series of families of gvisitor-like functions:

```
gvisitor2 :: InputIterator a b =>
    (Maybe b -> c -> d) -> (c -> d -> c) -> c -> a -> d
gvisitor2 f g k = gvisitor (\i j -> f i (foldl g k j))

gvisitor3 :: InputIterator a b =>
    (Maybe b -> c -> d) -> (d -> c -> c) -> c -> a -> d
gvisitor3 f g k = gvisitor (\i j -> f i (foldr g k j))

gvisitor4 :: InputIterator a b =>
    (b -> c -> c) -> (c -> c -> c) -> c -> a -> c
gvisitor4 f g k =
     gvisitor (\i j -> fapply2 f i (foldr (g) k j)

gvisitor5 :: InputIterator a b =>
    (b -> b -> b) -> b -> a -> b
gvisitor5 f k = gvisitor4 f f k
```

Visitors - Simple Examples

We can now give a few simple examples of generic functions over polynomial containers:

- gelems - Number of elements of a Container;

```
gelems :: InputIterator a b => a -> Int
gelems = gvisitor4 (\x y -> succ y) (+) 0
```

- gdepth - Depth of a Container;

```
gdepth :: InputIterator a b => a -> Int
gdepth = gvisitor3 (\x y -> succ y) max 0
```

- gsum - Sum of a numeric Container.

```
gsum :: (Num b,InputIterator a b) => a -> b
gsum = gvisitor5 (+) 0
```

polynomial Data-types - Iterators

Moreover, if we also have:

```
write<Γ,Ω> :: Φ -> Γ* -> Γ
```

The function write will construct a type Γ. This function is closely related to the constructors (as well as in_Γ).
One could define a function itmap (metamorphism):

```
itmap<Γ₁,Γ₂,Ω> :: (Φ₁-> Φ₂) -> Γ₁-> Γ₂
```

In Haskell, we would again use type classes:

```
class OutputIterator Γ Φ | Γ -> Φ where
    write :: Maybe Φ -> [Γ] -> Γ

itmap :: (InputIterator a b, OutputIterator c d) =>
         (b -> d) -> a -> c
itmap f = gvisitor (\j k -> write (apply f j) k)
    where apply f x = maybe Nothing (Just . f) x
```

Iterators - Simple Examples

- convert - Converts an InputIterator into an OutputIterator.

```
convert :: (InputIterator a b,OutputIterator c b) => a -> c
convert = itmap id
```

This is an overloaded function (generic function), so in order to use it, you must explicitly type the result.

```
> (convert atree) :: [Integer]
```

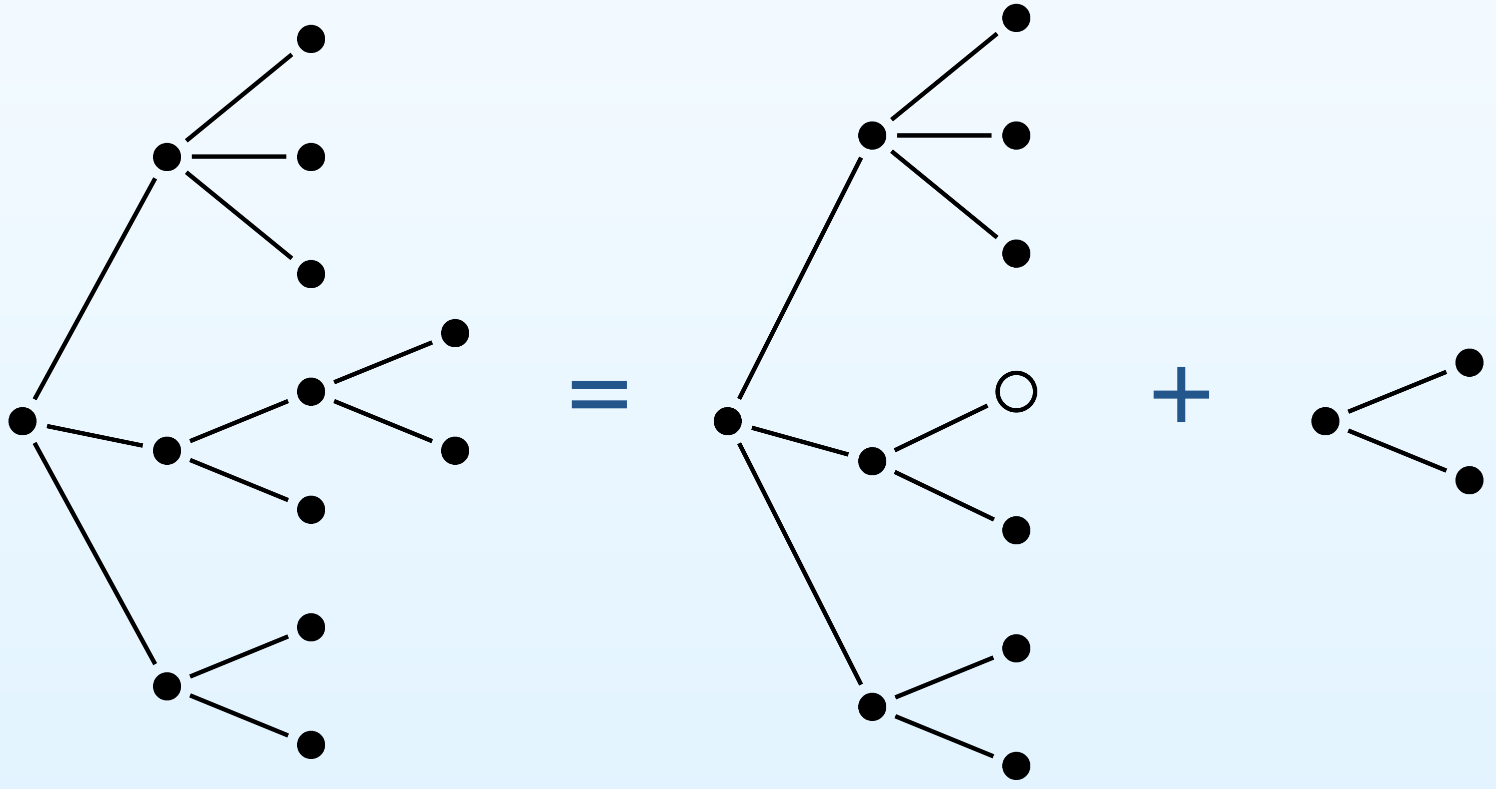
- flatten - Converts an InputIterator into a Haskell list.

```
flatten :: (InputIterator a b,OutputIterator [b] b) => a -> [b]
flatten = itmap id
```

In this case, you define the final type in the function signature, so you don’t need to explicitly type the result.

Zipper

Gerard Huet's paper "The Zipper" presents an elegant representation of tree like data, decomposed into the focus of the tree and its one-hole context. The focus of the tree is the current subtree that we are visiting, and the one-hole context is the tree with a hole in the position of the focus subtree.



Zipper

We can define a zipper for a polynomial data-type, using:

```
data Path a b = Top | Node [a] (Maybe b,Path a b) [a]

type Location a b = Maybe (a,Path a b)
```

With the functions value and succs defined for the *InputIterator* is possible define the functions down, left and right of the *Zipper*.

```
down (Just (t,p)) = case succs t of
    Nothing -> Nothing
    Just s -> Just (head s,Node [] (value t,p) (tail s))

left (Just (t,p)) = case p of
    Top -> Nothing
    Node [] _ _ -> Nothing
    Node (x:xs) u r -> Just (x,Node xs u (t:r))
```

Zipper

In order to be able to go up, we will need one extra generic function:

```
build<Γ,Ω> :: Γ -> Maybe Φ -> [Γ] -> [Γ] -> Γ
```

The function build will take, the current recursive element in the *Zipper*, plus the value for that element, the elements on its left and right and produce the element on top.
In Haskell, we could define the *Zipper* as:

```
class InputIterator Γ Φ => Zipper Γ Φ where
      build :: Γ -> Maybe Φ -> [Γ] -> [Γ] -> Γ
```

up would then be defined as:

```
up (Just (t,p)) = case p of
      Top -> Nothing
      Node l u r -> Just (build t
                     (fst u) (reverse l) r,snd u)
```

Proposed Features

- Data-type transformers (DTT) support:

$\theta :: Type \rightarrow Type$

- Generic Functions with support for DTT:

value<Γ,Ω> :: Γ -> Φ

- Multi-parameters:

write<Γ,Ω> :: Φ -> Γ^* -> Γ

Conclusions and future work

- Check for the feasibility of the proposed features;
- Explore possible syntax, as well as type checking issues for the features proposed;
- If feasible, implement them in a Generic Programming tool;
- Improve the theory presented, in order to be extensible to any data-type.