Exploring Lightweight Implementations of Generics

Bruno Oliveira
Introduction

**Generic Programming** is about defining functions that can work on a family of datatypes independently of their shape. A particular type of Generic Programming relies on structural polymorphism to achieve this goal. In this talk, we propose a slightly different approach to type representations: instead of having a set of constructors for the base cases, we will have a single constructor for any "constant" case. This constructor combined with a typeclass, allows greater flexibility. For instance, it is possible to define a function that works on any type that is an instance of a particular typeclass. This allows us to achieve this goal.

Typically, type representations are defined using a set of constructors for the structural cases (sums and products) and a set of constructors for the base cases (primitive types such as Int, Char, Bool, ...). In “Generics for the Masses”, we can have generic programming with a relatively modest type system. For instance, in “Generics for the Masses”, type representations can be used to simulate the behaviour of “typeclasses”. This allows us to define very general typeclasses, a particular type of Generic Programming that can work on a family of datatypes independently of their shape. A particular type of Generic Programming relies on structural polymorphism to achieve this goal.

In this talk, we propose a slightly different approach to type representations: instead of having a set of constructors for the base cases, we will have a single constructor for any "constant" case. This constructor combined with a typeclass, allows greater flexibility. For instance, it is possible to define a function that works on any type that is an instance of a particular typeclass. This allows us to achieve this goal.

Typically, type representations are defined using a set of constructors for the structural cases (sums and products) and a set of constructors for the base cases (primitive types such as Int, Char, Bool, ...). In “Generics for the Masses”, we can have generic programming with a relatively modest type system. For instance, in “Generics for the Masses”, type representations can be used to simulate the behaviour of “typeclasses”. This allows us to achieve this goal.
Parametric polymorphism allows us to express functions that work uniformly for all types.

Ad-hoc polymorphic functions allow us to define functions per type case.

$size = size_{Tree}$

instance $Size (Tree a)$ where

$size = size_{List}$

instance $Size [a]$ where

$size :: Size c \Leftarrow Int$

$size = size_{List} [a] :: Int$

$size :: Tree a \Leftarrow Int$

$size :: Int \Leftarrow [a] :: Int$
The function \( \text{size} \) is called a generic function.

\[
\begin{align*}
\text{size} + x \cdot \text{size} & = (\text{size} \cdot x) \\
\text{size} & = (\text{size} \cdot \text{Int}) \\
x \cdot \text{size} & = (x \cdot \text{size}) \\
0 & = \text{size} \cdot \text{Unit} \\
1 & = \text{size} \cdot \text{Char} \\
1 & = \text{size} \cdot \text{Int} \\
\text{Int} & \leftarrow \text{size} \cdot \text{Char} \\
\text{Char} & \leftarrow \text{size} \cdot \text{Int}
\end{align*}
\]

With Structural Polymorphism we can define functions over the structure of types.
In a lightweight approach to generics and dynamics (Ralf Hinze), we are shown how to encode type representations using existential types and an equality type.

```haskell
{ a ← q :: to, q ← a :: from } = q ≡ a

data Rept =
  RUnit (t :: Unit) | RInt (t :: Int) | RChar (t :: Char)

data Rep =
  Rep (q, Rep) (q, Rep) (q, Rep) (q, Rep) (q, Rep) (q, Rep)

data a b = f from :: a ! b; to :: b ! a g
```

Type Representations
Using type representations, generic functions are just normal functions.
An alternative approach is to replace the base case by a single “constant” case.

"Constant" case: a case where the behavior for any constant case can combine the constant case with a two-parameter type class, which allows us to define the specific function.

Type Representations and ad-hoc polymorphism

Bruno Oliveira
Now we can define \texttt{rSize} as:

\begin{align*}
0 \oplus \text{rFold rep} = \text{rSize rep} \oplus 0
\end{align*}

Defining a uniform generic fold function:

\begin{align*}
\text{rFold rep} \in f \mathrel{\text{fold in f x}} \in \text{fFold rep} \in f \mathrel{\text{f fold in f y}} \in \text{fFold rep} \in f = rFold rep \mathrel{\text{RUnitep fghkt1}} = k
\end{align*}

\begin{align*}
\text{rFold rep} \mathrel{\text{RConst ep fghkt1}} = \text{RConst ep} fghkt1
\end{align*}

\begin{align*}
\text{case (from ep \text{f}) of (}
\text{Prod x y fghkt1} = \text{Prod x y fghkt1}
\text{case (from ep \text{f}) of (}
\text{Inl x fghkt1} = \text{Inl x fghkt1}
\text{case (from ep \text{f}) of (}
\text{Inr y fghkt1} = \text{Inr y fghkt1}
\text{case (from ep \text{f}) of (}
\text{Const x baseFunc x fghkt1} = \text{Const x baseFunc x fghkt1}
\text{case (from ep \text{f}) of (}
\text{Prod xy fghkt1} = \text{Prod xy fghkt1}
\end{align*}

\begin{align*}
\text{rFold rep} \mathrel{\text{RSum rep fghkt1}} = \text{RSum rep fghkt1}
\end{align*}

\begin{align*}
\text{case (from ep \text{f}) of (}
\text{Prod xy fghkt1} = \text{Prod xy fghkt1}
\end{align*}

\begin{align*}
\text{rFold rep} \mathrel{\text{RProdr rep fghkt1}} = \text{RProdr rep fghkt1}
\end{align*}

\begin{align*}
\text{case (from ep \text{f}) of (}
\text{Prod xy fghkt1} = \text{Prod xy fghkt1}
\end{align*}

\begin{align*}
\text{rFold rep} \mathrel{\text{+ id id}} = \text{rSize rep} \oplus 0
\end{align*}

Now we can define \texttt{rSize} as:

\begin{align*}
\text{rSize rep} = \text{rFold rep} \mathrel{\text{+ id id}} 0
\end{align*}
In order to use datatypes, we need to provide the representation for each datatype.

---

**Embedding/Projecting Datatypes**

Exploring Lightweight Implementations of Generics
Exploring lightweight implementations of Generics

Method 1 - Local redefinitions

One possible language extension would allow two different methods to declare the base cases:

Local redefinitions would be particularly useful in the presence of (natively declared) default cases, allowing to override default behaviour with some more specific behaviour.

Method 2 - Converting base cases into higher-order functions

The syntax resembling a higher-order function \( gsize \) would permit a very compact syntax for calling the generic functions.

\[
\text{let } \text{gsize} = f \text{ in } \text{gsize} \langle \text{Int} \text{ const} \text{ I} \rangle = f
\]
Problems with typeclasses: the syntaxis lengthy; we might need extra extensions — overlapping, undecidable instances... — for more general base cases (ex. `BaseCase a`); type classes are

Advantages of type classes: they are already there; it is easy to overcome the problems, for instance, if not local.

One type representation is not enough to allow the definition of all generic functions. For instance, in...

The same technique, can be used with other similar approaches: Generics for the Masses (Hatt)

The type representation necessary to encode such function.

One type representation is not enough to allow the definition of all generic functions. For instance, in...

The same technique, can be used with other similar approaches: Generics for the Masses (Hatt)

Advantages of type classes: they are already there; it is easy to overcome the problems, for instance, if not local.

One type representation is not enough to allow the definition of all generic functions. For instance, in...

The type representation necessary to encode such function.

One type representation is not enough to allow the definition of all generic functions. For instance, in...

The same technique, can be used with other similar approaches: Generics for the Masses (Hatt)

Advantages of type classes: they are already there; it is easy to overcome the problems, for instance, if not local.

One type representation is not enough to allow the definition of all generic functions. For instance, in...

The type representation necessary to encode such function.

One type representation is not enough to allow the definition of all generic functions. For instance, in...

The same technique, can be used with other similar approaches: Generics for the Masses (Hatt)

Advantages of type classes: they are already there; it is easy to overcome the problems, for instance, if not local.

One type representation is not enough to allow the definition of all generic functions. For instance, in...

The type representation necessary to encode such function.

One type representation is not enough to allow the definition of all generic functions. For instance, in...

The same technique, can be used with other similar approaches: Generics for the Masses (Hatt)

Advantages of type classes: they are already there; it is easy to overcome the problems, for instance, if not local.

One type representation is not enough to allow the definition of all generic functions. For instance, in...

The type representation necessary to encode such function.