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.
- Type representations can be used to simulate the behaviour of “typecases”. This allows us to have generic programming with a relatively modest type system. For instance, in “Generics for the Masses”, we can have generic programming in Haskell 98.
- 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 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 type class, allows greater flexibility. For instance, it is possible to override the behaviour of a generic function definition at some specific instances.

Polymorphism

Parametric polymorphism allows us to express functions that work uniformly for all types.

$$sizeList :: [a] \quad \rightarrow Int$$
$$sizeTree :: Tree\ a \rightarrow Int$$

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

$$size :: Size\ c \Rightarrow c \rightarrow Int$$

$$\mathbf{instance}\ Size\ [a]\ \mathbf{where}$$
$$\quad size = sizeList$$

$$\mathbf{instance}\ Size\ (Tree\ a)\ \mathbf{where}$$
$$\quad size = sizeTree$$

Structural Polymorphism - Generic Programming

With *Structural polymorphism* we can define functions over the structure of types.

$$
\begin{array}{lll}
size\langle c\rangle :: & c & \rightarrow Int \\
size\langle Int\rangle & \_ & = 1 \\
size\langle Char\rangle & \_ & = 1 \\
size\langle Unit\rangle & \_ & = 0 \\
size\langle Sum\ a\ b\rangle & (Inl\ x) & = size\langle a\rangle\ x \\
size\langle Sum\ a\ b\rangle & (Inr\ y) & = size\langle b\rangle\ y \\
size\langle Prod\ a\ b\rangle & (x, y) & = size\langle a\rangle\ x + size\langle b\rangle\ y
\end{array}
$$

The function $size$ is called a *generic function*.

Type Representations

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.

$$
\begin{array}{ll}
\mathbf{data}\ Rep\ t = & \\
\quad\ \ RUnit & (t \equiv\ Unit) \\
\quad \mid RInt & (t \equiv\ Int) \\
\quad \mid RChar & (t \equiv\ Char) \\
\quad \mid \forall\ a\ b.RSum\ (Rep\ a)\ (Rep\ b) & (t \equiv\ (Sum\ a\ b)) \\
\quad \mid \forall\ a\ b.RProd\ (Rep\ a)\ (Rep\ b) & (t \equiv\ (Prod\ a\ b))
\end{array}
$$

$$\mathbf{data}\ a \equiv\ b = \{from :: a \rightarrow b, to :: b \rightarrow a\}$$

Type Representations

Using type representations, *generic functions* are just normal functions.

$$
\begin{array}{l}
rSize :: Rep\ t \rightarrow t \rightarrow Int \\
rSize\ (RUnit\ ep)\ \_ \qquad\quad = 0 \\
rSize\ (RInt\ ep)\quad t1 \qquad\quad = from\ ep\ t1 \\
rSize\ (RChar\ ep)\ \_ \qquad\quad = 0 \\
rSize\ (RSum\ ra\ rb\ ep)\ t1 = \\
\quad \mathbf{case}\ (from\ ep\ t1)\ \mathbf{of} \\
\qquad (Inl\ x) \rightarrow rSize\ ra\ x \\
\qquad (Inr\ x) \rightarrow rSize\ rb\ x \\
rSize\ (RProd\ ra\ rb\ ep)\ t1 = \\
\quad \mathbf{case}\ (from\ ep\ t1)\ \mathbf{of} \\
\qquad (Prod\ x\ y) \rightarrow rSize\ ra\ x + rSize\ rb\ y
\end{array}
$$

Type Representations and ad-hoc polymorphism

An alternative approach is to replace the base cases (Int, Char, ...) by a single ”constant” case. We can combine the constant case with a two-parameter type class, which allows us to define the specific behaviour for any constant case.

$$
\begin{array}{l}
\textbf{class}\ BaseCase\ a\ b\ \textbf{where} \\
\quad baseFunc :: a \rightarrow b \\
\textbf{data}\ Rep\ t\ u = \\
\quad RUnit \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (t \equiv\ Unit) \\
\quad \mid \forall\ b.BaseCase\ b\ u \Rightarrow RConst\ (Type\ b)\ (t \equiv\ (Const\ b)) \\
\quad \mid \forall\ a\ b.RSum\ (Rep\ a\ u)\ (Rep\ b\ u) \qquad (t \equiv\ (Sum\ a\ b)) \\
\quad \mid \forall\ a\ b.RProd\ (Rep\ a\ u)\ (Rep\ b\ u) \qquad (t \equiv\ (Prod\ a\ b))
\end{array}
$$

Type Representations and ad-hoc polymorphism

Defining a uniform generic $fold$ function.

$$
\begin{array}{l}
rFold :: Rep\ t\ a \rightarrow (a \rightarrow a \rightarrow a) \rightarrow (a \rightarrow a) \rightarrow (a \rightarrow a) \rightarrow a \rightarrow t \rightarrow a \\
rFold\ (RUnit\ ep) \qquad f\ g\ h\ k\ t1 = k \\
rFold\ (RConst\ \_\ ep) \quad f\ g\ h\ k\ t1 = \\
\quad \mathbf{case}\ (from\ ep\ t1)\ \mathbf{of} \\
\qquad Const\ x \rightarrow baseFunc\ x \\
rFold\ (RSum\ ra\ rb\ ep)\ f\ g\ h\ k\ t1 = \\
\quad \mathbf{case}\ (from\ ep\ t1)\ \mathbf{of} \\
\qquad Inl\ x \rightarrow g\ (rFold\ ra\ f\ g\ h\ k\ x) \\
\qquad Inr\ y \rightarrow h\ (rFold\ rb\ f\ g\ h\ k\ y) \\
rFold\ (RProd\ ra\ rb\ ep)\ f\ g\ h\ k\ t1 = \\
\quad \mathbf{case}\ (from\ ep\ t1)\ \mathbf{of} \\
\qquad Prod\ x\ y \rightarrow f\ (rFold\ ra\ f\ g\ h\ k\ x)\ (rFold\ rb\ f\ g\ h\ k\ y)
\end{array}
$$

Now we can define $rSize$ as:

$$rSize\ rep = rFold\ rep\ (+)\ id\ id\ 0$$

Embedding/Projecting Datatypes

In order to use datatypes, we need to provide the representation for each datatype.

$$rList \quad :: Rep\ (Const\ a)\ u \rightarrow Rep\ [a]\ u$$
$$rList\ ra = RSum\ rUnit\ (rProd\ ra\ (rList\ ra))\ (fromList \equiv\ toList)$$

In this case, $\equiv$ becomes an *isomorphism*.

$$fromList :: [a] \rightarrow Sum\ Unit\ (Prod\ (Const\ a)\ [a])$$
$$toList \quad :: Sum\ Unit\ (Prod\ (Const\ a)\ [a]) \rightarrow [a]$$

A session:

$> \mathbf{let}\ repL = rList\ (rConst\ (Type :: Type\ a))$

$> :t\ repL$

$\quad rList\ (rConst\ (Type :: Type\ a)) :: \forall\ u.(BaseCase\ a\ u) \Rightarrow Rep\ [a]\ u$

$> rSize\ repL\ [1\,..\,10]$

$No\ \mathbf{instance}\ for\ (BaseCase\ Int\ Int)$

$> \mathbf{let\ instance}\ BaseCase\ Int\ Int\ \mathbf{where}\ baseFunc\ x = 1$ -- Pseudo code

$> rSize\ repL\ [1\,..\,10]$

10

A language extension

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

Method 1 - Local redefinitions

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

$$\begin{aligned} f = \mathbf{let}\ & gsize\langle Int\rangle = const\ 1 \\ \mathbf{in}\ & gsize\ [1\,..\,10] \end{aligned}$$

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.

$$f = gsize\ (const\ 1)\ [1\,..\,10]$$

Conclusions

- Problems with type classes: the *syntax is lengthy*; we *might need extra extensions* — overlapping, undecidable instances . . . — for more general base cases (ex. $BaseCase\ a\ a$); type classes are *not local*.
- Advantages of type classes: they are *already there*; it is easy to overcome the problems, for instances, it is possible to simulate the behaviour of local instances using different modules.
- One type representation is not enough to allow the definition of *all* generic functions. For instance, it is possible to encode a polymorphic map, but we require a slightly different representation. *However,* we do not think this is a problem: there is a direct relation between the type of a generic function and the type representation necessary to encode such function.
- The same technique, can be used with other similar approaches: *Generics for the Masses* (Ralf Hinze); GADTS; universal types. For instance, we can combine this technique with Generics for the Masses and still remain in *Haskell 98*.
- There is a big similarity between *Dependency-style Generic Haskell* and this approach. We believe that a translation from Dependency-style Generic Haskell to Haskell using this approach is possible. This is future work.