Objects to Unify Type Classes and GADTs

Bruno Oliveira (Speaker)
Oxford University Computing Laboratory
bruno@comlab.ox.ac.uk

Martin Sulzmann
School of Computing, National University of Singapore
sulzmann@comp.nus.edu.sg

Introduction

Type classes and GADTs have clear (syntactical) similarities; but the two mechanisms seem somehow orthogonal. Some common questions:

- Given some problem how can we decompose it? Common Haskeller question: shall I use datatypes or shall I use type classes?
- What is the relationship between datatypes, type classes and OO-classes? Common newbie (but with OO experience) Haskeller question: how do I translate my OO program into Haskell?

Related main question addressed in this talk:

> Can we have a single type of declaration that allows us to unify and generalize type classes and GADTs?

The Set Problem

Define an ADT for Sets in Haskell that:

1. supports multiple implementations;
2. is (obviously!) abstract —that is, the concrete implementation(s) is/are hidden;
3. allows us to dynamically replace one implementation by another one.

Different Solutions for the Set Problem in Haskell

Haskell is a rich language with many different constructs and features. Here are a few solutions that we could try:

1. Modules —This is how Sets are implemented in the Haskell libraries.
2. Existential types —We can use existential types to encode ADTs.
3. Type Classes (Bulk Types) —We can use type classes to support multiple implementations.
4. Records —We can encode objects with records.

Haskell does offer multiple choices, but each of them is somewhat unsatisfactory since it either fails to meet all the requirements or it is not very straightforward.

A Straightforward OO Solution

Imagine yourself as being an experienced OO programmer trying to learn Haskell; and you have just come across the concept of type classes. You may be tempted to write:

$$\begin{aligned}&\textbf{class } Set\ a \textbf{ where}\\ &\quad member :: a \to Bool\\ &\quad insert\ \ :: a \to Set\ a\end{aligned}$$

Unfortunately, this is rejected in Haskell:

> Class 'Set' used as a type
> In the class declaration for 'Set'

INTRODUCING H+-

H+- is more or less Haskell:

H+- $\approx$ Haskell + objects - a number of other features

H+- features:

- a generalized class system where type class instances are replaced by the more general concept of objects;
- a powerful dot notation which allows us to easily switch between implicitly passed and explicitly passed arguments;
- sealed classes: all objects that construct values of a certain class are defined in the same module as that class (closed world perspective).
- open GADTs and closed GADTs with pattern matching by interpreting classes as datatypes and objects as value constructors.

A Solution for the Set Problem in H+-

Unlike in Haskell, H+- classes can occur in type positions, so the OO solution is a valid program:

$$\begin{aligned}
&\textbf{class } Set\ a \textbf{ where}\\
&\quad member :: a \to Bool\\
&\quad insert\ \ :: a \to Set\ a
\end{aligned}$$

Here is a possible implementation:

$$\begin{aligned}
&\textbf{object } ListSet :: Ord\ a \Rightarrow [a] \to Set\ a\\
&\textbf{object } ListSet\ xs \textbf{ where}\\
&\quad member\ x = elem\ x\ xs\\
&\quad insert\ x\ \ = ListSet\ (union\ [x]\ xs)
\end{aligned}$$

Constructing object values and the Dot notation

Like with Haskell type classes, the method $insert$ has the type:

$$insert :: Set\ a \Rightarrow a \rightarrow Set\ a$$

However, in H+- we can explicitly pass a value for the dictionary argument by using the dot notation.

$$ins :: Set\ a \rightarrow a \rightarrow Set\ a$$

$$ins\ s\ x = s.insert\ x$$

$$set :: Set\ Int$$

$$set = ins\ (ins\ (ins\ (ListSet\ [])\ 3)\ 4)\ 3$$

The value set shows how we can use a value constructor-like notation to build objects.

Application: Open Datatypes

We can interpret an H+- class and its objects as a form of open datatype. For example, suppose that we want to define the $sprintf$ function in H+- and we want to be able to add new kinds of format specifiers in the future.

Ideally, we would like to have an extensible $Format$ datatype. Here is how we can write the $Format$ type and the $sprintf$ function in H+-:

$$\begin{array}{l}\textbf{class } Format\ t \textbf{ where} \\ \quad sprintf' :: String \to t \\ sprintf :: Format\ t \to t \\ sprintf\ f = f.sprintf'\ ""\end{array}$$

Application: Open Datatypes

A few format specifiers:

$$\textbf{instance object } E \textbf{ where} \quad \text{-- } E :: Format\ String$$
$$\quad sprintf' = id$$

$$\textbf{instance object } I :: Format\ t \Rightarrow Format\ (Int \rightarrow t)$$

$$\textbf{instance object } I \textbf{ where}$$
$$\quad sprintf'\ s\ x = sprintf'\ (s +\!\!+ show\ x)$$

$$\textbf{object } S\ x\ k \textbf{ where} \quad \text{-- } S :: String \rightarrow Format\ t \rightarrow Format\ t$$
$$\quad sprintf'\ s = k.sprintf'\ (s +\!\!+ x)$$

A few things to note: instance objects can be used in the same way as type class instances in Haskell; the type signature of an object is optional and can often be inferred; and the types of the object constructors are just like GADT value constructors.

Application: Implicit Explicit Objects

A common problem in Haskell, derived from the fact that we cannot explicit pass `dictionaries', is the existence of 'By' functions:

$$
\begin{aligned}
&insert &&:: Ord\ a \Rightarrow a \rightarrow [a] \rightarrow [a] \\
&insertBy &&:: (a \rightarrow a \rightarrow Bool) \rightarrow a \rightarrow [a] \rightarrow [a] \\
&sort &&:: Ord\ a \Rightarrow [a] \rightarrow [a] \\
&sortBy &&:: (a \rightarrow a \rightarrow Bool) \rightarrow [a] \rightarrow [a]
\end{aligned}
$$

In H+- we can write:

$$
\begin{aligned}
&insert &&:: Ord\ a \Rightarrow a \rightarrow [a] \rightarrow [a] \\
&insert\ x\ [] &&= [x] \\
&insert\ x\ (y : ys) &&= \mathbf{if}\ x > y\ \mathbf{then}\ y : insert\ x\ ys\ \mathbf{else}\ x : y : ys
\end{aligned}
$$

$$
\begin{aligned}
&sort :: Ord\ a \Rightarrow [a] \rightarrow [a] \\
&sort = foldr\ insert\ []
\end{aligned}
$$

Application: Implicit Explicit Objects

We can use $sort$ like we would do in Haskell:

$$sortImplicit :: [Int]$$
$$sortImplicit = sort\,[2, 6, 5]$$

But, using the . notation, we can also override the Ord dictionary and provide our own:

$$\textbf{object}\ OrdTrue :: Ord\ Int$$
$$\textbf{object}\ OrdTrue\ \textbf{where}$$
$$\quad x > y = true$$
$$sortExplicit :: [Int]$$
$$sortExplicit = OrdTrue.sort\,[2, 6, 5]$$

Sealed Classes

The H+- classes that we have seen so far are always open: we can always add new objects (even if they are in a different module). However, we cannot add new methods or functions defined by pattern matching.

With sealed classes new instances/objects cannot be declared outside the module that defines the class but we gain, in return, the ability to do pattern matching. For example:

$$\begin{aligned}
&\textbf{sealed class } \mathit{Exp}\ a \textbf{ where}\\
&\textbf{object } \mathit{Lit}\ \ :: \mathit{Int} \to \mathit{Exp}\ \mathit{Int}\\
&\textbf{object } \mathit{Plus} :: \mathit{Exp}\ \mathit{Int} \to \mathit{Exp}\ \mathit{Int} \to \mathit{Exp}\ \mathit{Int}\\
&\textbf{object } \mathit{IsZ}\ \ :: \mathit{Exp}\ \mathit{Int} \to \mathit{Exp}\ \mathit{Bool}\\
&\textbf{object } \mathit{If}\ \ \ :: \mathit{Exp}\ \mathit{Bool} \to \mathit{Exp}\ a \to \mathit{Exp}\ a \to \mathit{Exp}\ a
\end{aligned}$$

Sealed Classes

Now we can easily define new functions by pattern matching on the sealed class Exp:

$$
\begin{aligned}
&eval && :: Exp\ a \to a \\
&eval\ (Lit\ x) && = x \\
&eval\ (Plus\ e1\ e2) && = eval\ e1 + eval\ e2 \\
&eval\ (IsZ\ e) && = eval\ e \equiv 0 \\
&eval\ (If\ p\ e1\ e2) && = \mathbf{if}\ eval\ p\ \mathbf{then}\ eval\ e1\ \mathbf{else}\ eval\ e2
\end{aligned}
$$

Sealed Classes and Methods

We have seen how to use sealed classes to define GADTs, but classes are more general since they can have methods. What does it mean to have a GADT with methods?

Methods are attributes of the class and not of the objects. Suppose we want to improve the messages reported to the user by adding extra location information to our expressions. We can do this with methods:

$$\textbf{type } Loc = Maybe\ (Int, Int)$$

$$\textbf{sealed class } Exp\ a \textbf{ where}$$

$$loc :: Loc$$

$$loc = Nothing$$

Sealed Classes and Methods

Now suppose we add support for division:

$$\begin{aligned}
&\textbf{sealed class } \mathit{Exp}\ a \textbf{ where}\\
&\ldots\\
&\textbf{object } \mathit{Div} :: \mathit{Exp}\ \mathit{Int} \to \mathit{Exp}\ \mathit{Int} \to \mathit{Exp}\ \mathit{Int}\\
&\mathit{eval} :: \mathit{Exp}\ a \to a\\
&\ldots\\
&\mathit{eval}\ d@(\mathit{Div}\ e1\ e2) = \textbf{if } \mathit{eval}\ e2 \not\equiv 0 \textbf{ then } \mathit{eval}\ e1\ `\mathit{div}`\ \mathit{eval}\ e2\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } \mathit{error}\ (\mathit{show}\ d.\mathit{loc} +\!\!+ \texttt{": division by 0"})
\end{aligned}$$

The thing to note is that we basically only need to modify code that makes use of the extra location information.

Conclusions

- GADTs and type classes blend nicely together thanks to objects.
- H+- classes are first-class because we allow explicit dictionary-passing in the source language.
- Hopefully this work will help answering questions such as: "how do I translate my OO program into an Haskell program?" or "should I use type classes or datatypes for this problem?".
- Non-goal - We are not aiming at a Haskell backwards compatible extension; instead we want to devise the simplest system possible (tuple vs set semantics on constraints).
- Issue - Context reduction vs late binding.