The Monad.Reader/Issue3/Functional Programming vs Object Oriented Programming

From HaskellWiki

< The Monad.Reader | Issue3

This article arose from a discussion I had with Shae Erisson (shapr) after he witnessed some drivel I left on one of the c2.com wiki pages. It's not really a discussion of the pros and cons of functional or object-oriented programming. Rather, it's a guide for a new Haskell programmer coming from languages like Java, C#/.Net, or even Pascal or C, and intends to give a mapping from OO concepts to Haskell concepts.

Contents

1 FP <-> Immutable Objects

A Haskell program is like an OO program where every object is immutable. All you can do is construct new objects (this makes the term "data constructor" seem more relevant). When you want to change an object, you make a copy of it and replace just the bits you want to change. This isn't as expensive as it sounds, due to the extensive data sharing that immutable objects make possible (the bits of the object that aren't modified aren't copied; you just retain the references to the originals). Also, you avoid any aliasing problems, which is a potential cause of errors in programs that use mutable state.

Haskell also has no notion of object identity. When you construct a value (an instance of a type), the constructor creates a box (pointer) around the value. When you pass this value to functions, you pass the box, so it could be considered a reference. But Haskell has no way of saying "are these the same object?" i.e. are these the same box? All it has is an equality operator (==), which is only concerned with the box contents.

Object systems do have a notion of object identity. This seems to go hand-in-hand with mutable state, because the questions "are these the same object" and "are these the same value" are both relevant.

When you have immutable objects, the question "are these the same object" seems a bit pointless, whereas "are these the same value" is always relevant. "Are these the same object" isn't interesting because the behaviour of your program can't vary with the answer, whereas with mutable state, the behaviour of your program can vary with the answer (this is aliasing).

If you have mutable state then you can't *guarantee* referential transparency. That's the big deal with Haskell: no mutable state means you *can* guarantee referential transparency, and that guarantee is a Very Good Thing (all sorts of benefits flow from this).

Amr Sabry gives a definition of purity: http://citeseer.ist.psu.edu/sabry98what.html

2 FP <-> Higher Order Functions

The Haskell (and FP) style makes heavy use of higher-order functions (combinators), and parametric polymorphism. I think that higher-order functions are the defining characteristic of FP. Note that some OO languages (Ruby, Smalltalk, and Python, I think) support higher-order functions, so it's not an exclusive thing.

- "closures are a poor man's objects", "objects are a poor man's closures". [1] (http://www.ai.mit.edu/~gregs/ll1-discuss-archive-html/msg03277.html)

3 OO <-> Inclusional Polymorphism

A.K.A sub-typing. The OO style makes heavy use of inclusional polymorphism. IMHO, this is the defining characteristic of OO programming.

My biggest mistake (misconception) with Haskell was thinking that type classes were analogous to OO classes. Because there's a mechanism that looks a bit like inheritance (e.g. class (Eq a) => Ord a), I assumed that Haskell supported OO-style (inclusional) polymorphism.

The misunderstanding isn't helped by the this paper's references to OO subtyping: http://homepages.inf.ed.ac.uk/wadler/papers/class/class.ps

or the tail of this page, which compares Haskell to OO: http://www.haskell.org/tutorial/classes.html

4 Type Classes are not Types

Quite often an analogy is made with Haskell's type classes and OO interfaces (as you'd see in Java or C#). For example:

```
f :: MyInf a => a -> Integer
f x = ...
```

This is a bit like (pseudo-OO code):

```
interface MyInf { ... }

Integer f (x MyInf) { ... }
```

The big difference, of course, is that type classes are not types, whereas interfaces are (in the OO-version of function f, the type of x is MyInf).

5 OO and FP Styles Contrasted

There's an excellent comparison in the "Subtyping and polymorphism" section of this page: http://www.cs.chalmers.se/~nordland/ohaskell/rationale.html#subpoly

To quote:

> "The functional approach to programming is to ask "how is data constructed?". This leads to a style of programming where the data constructors are considered primitive, and where data-consuming code accordingly is defined by pattern-matching over these constructors. The object-oriented programmer instead starts out by asking "what can we do with the data?", which fosters the view that it is the data selectors that should be seen as foundational, and that the code of a data-producer consequently should be defined as an enumeration of implementations for these selectors."

What's telling is the description of the effects (local and non-local) of different types of modification to the program:

> "The functional style ensures that adding another data-consumer (a function) is a strictly local change, whereas adding a new data-producer requires extending a datatype with a new constructor, and thus a major overhaul of every pattern-matching function definition. Likewise, in object-oriented programming the addition of a new data-producer (a new class) is a cinch, while the introduction of another data-consumer means the extension of a record type with a new selector, a global undertaking with implications to potentially all existing classes."

So we have:

- FP data producer - datatype constructor
- FP data consumer - function
- OO data producer - class
- OO data consumer - class method

At first I had trouble imagining how adding a method to a class could potentially be a global undertaking, but then realised that if a new method was abstract, then you would have to visit every inherited class to add the same method.

6 I'm an OO programmer - how do I design Haskell programs?

Note that trying to design a Haskell program as you would an OO program is probably a mistake. It is likely that designing your program in "the Haskell style" is likely to result in a simpler program, and your code will probably look more like idiomatic Haskell, which is desirable from a maintenance point-of-view i.e. other Haskell programmers will find it easier to understand.

That said, it might be that you need to interface with external OO code, or you are porting an existing program and need to remain faithful to the original design, if only because you don't understand it well enough to convert it to a functional design.

- Q: I'm an OO programmer. If we don't have sub-typing/inheritance, how can I get polymorphic behaviour from functions? i.e. how do I get the equivalent of virtual functions/dynamic dispatch?
- A: These techniques:
  1. type classes
  2. using function types in (say) a data-type (e.g. this is used in HSQL to create a uniform interface for DBMS APIs)
  3. existentials
  4. use the OOHaskell code

Type classes, while giving polymorphic behaviour, are not OO. OO dispatch is value-based, where as type class dispatch is type-based. There are pros and cons to using type-classes; for example, see item 10 in: The Monad.Reader/Issue3/Notes on Learning Haskell#type-class-misuse The executive summary: don't think that you always need to use type classes.

The functions-in-a-datatype approach is a bit like constructing your own type-class dictionary, and might be more "natural" to an OO programmer who is familiar with the idea of bundling methods and data together. Also, you can implement value-based dispatch. The downside is that you're starting to roll-your-own OO system, and you can't easily extend subtypes by adding methods to them (i.e. your API is fixed).

I haven't used existentials at all, but they look promising. However, you're moving into the world of non-H98 extensions.

There's an excellent (i.e. quite readable) paper on mapping OO code to Haskell, although it's more from the point-of-view of interfacing to existing OO code i.e. via FFI: http://research.microsoft.com/Users/simonpj/Papers/oo-haskell/overloading.pdf

Some ideas in that paper are illustrated here: http://www.haskell.org/pipermail/haskell/2005-February/015346.html (the code referred to in that message follows). It looks like you can do OO-programming with type classes, but in fact the example is just a teaser. For example, you can't easily create the equivalent of a factory function which returns values in class Interpret (this is a consequence of the type-based, rather than value-based, dispatching mechanism):

```
type Id = String

-- Class for expression syntax
class Exp x

-- Expression syntax so far; could be extended
```

```
data Zero = Zero
data Exp x => Succ x = Succ x
data (Exp x, Exp y) => Then x y = Then x y
data Var = Var Id
data Exp x => Assign x = Assign Id x

instance Exp Zero
instance Exp x => Exp (Succ x)
instance (Exp x, Exp y) => Exp (Then x y)
instance Exp Var
instance Exp x => Exp (Assign x)

-- Semantic domain
type Val = Int

-- The monadic interpreter function

class Exp x => Interpret x where
 interpret :: x -> State (Map Id Val) Val

instance Interpret Zero where
 interpret Zero = return 0

instance Interpret x => Interpret (Succ x) where
 interpret (Succ x) = interpret x >>= return . (+) 1

instance (Interpret x, Interpret y) => Interpret (Then x y) where
 interpret (Then x y) = interpret x >>= const (interpret y)

instance Interpret Var where
 interpret (Var i) = get >>= return . fromJust . Data.Map.lookup i

instance Interpret x => Interpret (Assign x) where
 interpret (Assign i x) =
 do
 v <- interpret x
 m <- get
 let m' = insert i v m
 () <- put m'
 return v
```

The good news is that Oleg Kiselyov, Ralf Lämmel, and Keean Schupke have shown us how to build an object-oriented programming system in extended Haskell (multi-parameter type classes with functional dependencies): http://homepages.cwi.nl/~ralf/OOHaskell/ [2] (http://darcs.haskell.org/OOHaskell/)

Instead of requiring functional dependencies it has also been shown (https://github.com/UU-ComputerScience/js-asteroids/raw/master/msc-thesis/thesis.pdf) (Chapter 5) that a more primitive approach based on OOHaskell's mutable objects with tail-polymorphism approach suffices to model some of the basic OO features. The resulting library (https://github.com/UU-ComputerScience/js-asteroids/tree/master/lightoo) is used to implement a subset of wxWidgets (http://uu-computerscience.github.com/js-asteroids/) (C++) in terms of Haskell and Javascript.

Alistair Bayley

Retrieved from "http://www.haskell.org/haskellwiki/The_Monad.Reader/Issue3/Functional_Programming_vs_Object_Oriented_Programming"

Category: Article