Assignment 3

Hendrik Werner s4549775

April 17, 2017

1 Paper

I chose the paper "Data types la carte":

JFP **18** (4): 423–436, 2008.
doi:10.1017/S0956796808006758 First published online 18 March 2008 Printed in the United Kingdom

FUNCTIONAL PEARL

Data types à la carte

WOUTER SWIERSTRA

School of Computer Science, University of Nottingham, Jubilee Campus, Nottingham, NG8 1BB
(*e-mail:* wss@cs.nott.ac.uk)

Abstract

This paper describes a technique for assembling both data types and functions from isolated individual components. We also explore how the same technology can be used to combine free monads and, as a result, structure Haskell's monolithic IO monad.

1 Introduction

Implementing an evaluator for simple arithmetic expressions in Haskell is entirely straightforward.

```
data Expr = Val Int | Add Expr Expr

eval :: Expr → Int
eval (Val x)   = x
eval (Add x y) = eval x + eval y
```

Once we have chosen our data type, we are free to define new functions over expressions. For instance, we might want to render an expression as a string:

```
render :: Expr → String
render (Val x)   = show x
render (Add x y) = "(" ++ render x ++ " + " ++ render y ++ ")"
```

If we want to add new operators to our expression language, such as multiplication, we are on a bit of a sticky wicket. While we could extend our data type for expressions, this will require additional cases for the functions we have defined so far. Phil Wadler (1998) has dubbed this in the *Expression Problem*:

> The goal is to define a data type by cases, where one can add new cases to the data type and new functions over the data type, without recompiling existing code, and while retaining static type safety.

As the above example illustrates, Haskell can cope quite nicely with new function definitions; adding new constructors, however, forces us to modify existing code.

In this paper, we will examine one way to address the Expression Problem in Haskell. Using the techniques we present, you can define data types, functions, and even certain monads in a modular fashion.

2 Target Audience

The article is targeted at advances functional programmers and language designers, especially those familiar with Haskell.

The author uses advanced (functional) programming terms without explanation (monolithic, monads, ...). Also the article's topic is not generic, so that people not familiar with the matter will not be able to understand the content, nor would it be useful to them if they did.

3 Message

The article presents methods for solving the Expression Problem. To this end the author first introduced the Expression Problem, explains why it is indeed a problem, and then presents a solution to this problem. He then discusses the implications and importance of his approach.

4 Effectiveness of the Introduction

The introduction does a good job of presenting the Expression Problem and telling the reader what the rest of the paper is trying to achieve. There is even a short example program given, illustrating the problem, which I really like. This takes away much of the abstractness of the topic. The example is small, self contained, and easy to understand.