An Analysis and Discussion of Solutions to the Expression Problem Across Programming Languages

KEVIN TIAN, Stanford University
COLIN WEI, Stanford University

The "expression problem" in programming language design refers to the following phenomenon that occurs in naive implementations in functional or object-oriented programming languages: suppose we have a set of object classes, and a set of methods that the classes support. Then, while requiring that existing code not be modified, it is often simple to extend the code to include either an additional method (to be supported by all existing classes), or an additional class (to support all existing functions), but not both. Various solutions to this problem have been proposed in different programming languages, each with their unique sets of tradeoffs and features. In this paper we aim to provide a unified discussion and analysis on case studies of these solutions in Java, C++, Clojure, and OCaml. We also provide a novel solution which arises naturally in object-oriented languages by considering the duality of functions and objects in the expression problem. Our implementations can be found at https://github.com/cwein3/CS242Proj/.

Additional Key Words and Phrases: Expression problem, functional programming language, object oriented programming language

ACM Reference Format:
Kevin Tian and Colin Wei. 2017. An Analysis and Discussion of Solutions to the Expression Problem Across Programming Languages. 1, 1 (December 2017), 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The expression problem is now a classical problem in program language design, because in many ways it captures the expressibility of a language. The problem is as follows: consider a chunk of code that consists of some m "object classes" and n "functions", such that each object class should support its own implementation of each function. A simple example that we will use throughout this survey (and will call the "shapes example") is the situation where the object classes are different types of shapes (for example, Triangle, Square, Hexagon, ...) and the functions are computations which are specific to the type of shape (for example, angle, area, or perimeter calculation). The classic difficulty described by the expression problem arises when we try to extend the code to include either 1) an additional class which supports all existing functions, or 2) an additional method which is supported by all existing classes, both without modifying or duplicating existing code. The problem can be motivated by considering the following: clearly, if we are to add an additional function, we must write at least m pieces of additional code to describe the implementation for each existing class (and similar for adding classes); a solution to the expression problem must do the minimal amount of additional implementation in both of these cases.

The reason this problem is typically difficult is because most programming languages make one of these cases easy, and the other hard. For example, in functional languages, adding functions is easy, but adding classes is difficult –

Authors' addresses: Kevin Tian, Stanford University, kjtian@stanford.edu; Colin Wei, Stanford University, colinwei@stanford.edu.

naively, it requires retroactively modifying existing functions. Conversely, in object-oriented languages, adding objects is easy, but it is difficult to add a function without touching existing code. Since the problem has been proposed, there have been many workaround solutions that were discovered, which vary in simplicity, safety, and performance overhead. We aim to provide a survey of these methods in four languages, which we hope will serve as comprehensive case studies and illustrate why this problem is not as intimidating as it may appear. As far as we can tell, solutions fall under one of four general frameworks: object algebras, visitors, type classes, and multi-methods. We provide implementations of solutions falling under these frameworks in programming language settings that we found illuminating, benchmark the solutions, and then conclude by suggesting a novel approach and giving a discussion.

2 OBJECT ALGEBRAS

Oliveira and Cook [2012] introduce the concept of object algebras in order to solve the expression problem. According to Oliveira and Cook [2012], the advantage of object algebras is that they can be easily implemented in commonly used object-oriented languages such as Java and C#. The object algebras framework is relatively simple and does not require complex typing features.

Consider the shapes example. To implement object algebras, we would first define a `ShapeAlg` interface, which defines a function that implements the desired behavior for each type of shape: `triangle()`, `square()`, etc. Next, "factories" for different operations will implement this interface. For example, to implement the area computation, we would first define an `Area` interface which has a single function, `area()`. Then we could define an `AreaFactory` that implements `ShapeAlg` and constructs an `Area` object that implements `area()` in a manner that is specific to the shape type.

The expression problem is now solved because to add new shape types, we can simply extend the `ShapeAlg` interface to support a new function for that shape type. For example, to add support for hexagons, we can extend `ShapeAlg` to the `ShapeAlgWithHexagon` interface, which also includes a `hexagon()` function. We would then extend the already-existing factories to implement the new `hexagon()` function. Adding a new operation is easy to: we simply create a new factory for that operation. For example, to add the perimeter operation, we would create a `Perimeter` interface of objects which include the perimeter function, and then we would create a `PerimeterFactory` which creates `Perimeter` objects.

We implement our shapes setting using object algebras in order to provide a qualitative and quantitative evaluation of this solution to the expression problem. We use Java for our implementation. In our implementation, we explore both aspects of extensibility: we add a new angle calculation operation on top of the already existing area and perimeter operations, and we add a new hexagon shape. We then benchmark our code as follows: we construct 1000000 random shapes, and then perform our area, perimeter, and angle computations for each of these random shapes. This is the benchmark that will be used throughout the rest of the paper to test performance. For the object algebras setting, we compare against a standard object oriented solution.

Qualitatively, we find that the object algebras framework is indeed very easy to use once one understands it. In particular, the implementation did not require an advanced knowledge of extensibility and object oriented principles beyond the basic understanding that one could gain from an introductory computer science class. Object algebras also did not require any advanced object oriented features to implement.

We do note that object algebras make certain aspects of object oriented programming more unwieldy, however. For the pure object oriented implementation, in our benchmarks we simply created a `Triangle`, `Square`, etc. object depending on the shape type. Then to perform calculations on our shapes, we simply called `shape.CalcArea()`,

	Object Algebras	Object Oriented
Runtime (seconds)	0.327	0.139

Fig. 1. We show the average runtime, in seconds, of our shapes benchmark for both the object algebras and object oriented implementations. Times are averaged over 10 trials in order to reduce variance.

`shape.CalcPerimeter()`, and so on, without having to worry about the specific *type* of shape. However, since object algebra implementation did not have a notion of a shape object, we had to make several type-specific function calls for each shape. For example, to create a triangle, we had to call `AreaFactory.triangle()`, `PerimeterFactory.triangle()`, `AngleFactory.triangle()`. Perhaps there is a more clever way to circumvent this issue, but the object oriented approach certainly seems more natural and simpler in this case. This could become a hassle in more complicated projects that need to support multiple different operation types at once.

Quantitatively, we also found that the object algebras implementation suffered runtime losses compared to the object oriented approach. The object algebras implementation took around 2.5 times longer to perform the same operations. We hypothesize that this might be because the object algebras implementation is required to make a separate object for computing area, perimeter, and angle, instead of just a single shape object.

3 VISITOR FRAMEWORK

The visitor pattern [Krishnamurthi et al. 1998] is an older solution to the expression problem than object algebras, and object algebras are similar to the visitor pattern in many ways. The visitor pattern applies functional programming ideas to solve the expression problem in object-oriented languages. Both the visitor pattern and object algebras invert the object oriented paradigm by defining objects for the different operations we wish to perform and then defining functions corresponding to each type. However, object algebras claim to remove some of the overhead involved in the visitor pattern, namely the need for "accept" methods, while the visitor pattern seems more object-oriented in nature.

To implement the visitor pattern in the shapes setting, we first define a `ShapesVisitor` interface that has functions `VisitTriangle()`, `VisitSquare()`, etc., for each type of shape. These visit functions will take in a shape object and perform the desired operation on that object. For example, to implement area computation, we could define the `AreaVisitor` class which will implement VisitTriangle(), etc., to correctly compute the area for the specific shape. This idea is very similar to the approach taken by object algebras; however, the visitor pattern differs in the sense that we also need to define a `Shapes` interface with the function `Accept`. The purpose of the `Accept` function is to take in a visitor and call the visitor's operation on itself: for example, a `Triangle` class would implement the `Accept` function by taking in a generic `ShapesVisitor` and calling `VisitTriangle()` on itself. In a way, the visitor framework implements functional programming in an object oriented language by mapping different operations to various function objects.

To create new types in the visitor framework, we would again extend the shapes visitor interface to account for the additional type. For example, to add hexagons, we would extend `ShapesVisitor` to the `ShapesVisitorWithHexagon` interface, which contains a `VisitHexagon()` function. To add new operations, we could simply write a new visitor.

We implement the visitor framework in C++, defining the shapes family and area, perimeter, and angle operations. We benchmark our code in the same manner as our object algebra implementation: we generate 1000000 random shapes and then perform all three operations for each shape. For this experiment, we have three different implementations: a standard object oriented implementation, an inextensible visitor framework implementation, and an extensible visitor framework implementation. The inextensible visitor framework supports the addition of new operations, but does not

	Object Oriented	Inextensible Visitor	Extensible Visitor
Runtime (seconds)	0.142	0.211	0.301

Fig. 2. We show the average runtime, in seconds, of our shapes benchmark for the object oriented and visitor pattern implementations. Times are averaged over 10 trials in order to reduce variance.

have the object oriented programming bells and whistles needed to support additional types. Our implementations follow the outline provided in [Bendersky 2016].

We first provide our qualitative evaluation of the visitor framework. The visitor framework does seem to be more cumbersome to implement than standard object oriented programming, as it is also more counterintuitive (in our opinions) than the object algebras framework. However, because it is also more object-oriented than the object algebras framework, it avoided the problem with the object algebras framework discussed earlier. In particular, we did not need to call several different functions everytime we created a new shape. We instead could create a single `Shape` object each time, and when actually performing computations, call `shape -> Accept(areavisitor)`, etc. This gave a minor ease-of-implementation advantage over object algebras.

Quantitatively, we found that the visitor pattern performed worse than standard object oriented programming in terms of runtime. Again, we attribute this to increased overhead because the visitor pattern requires us to pass more objects around, thus requiring more operations.

4 TYPE CLASSES

Type classes are a very lightweight solution to the expression problem in staticly typed languages. The specific case study we consider is implementing polymorphic variants in OCaml, a functional language. We follow a solution outlined in [Kalmbach 2016].

Recall that the difficulty described by the expression problem in functional languages is that when we try to add an additional class, we cannot modify existing functions to handle this new case. A simple work-around would look as follows: suppose we have an interface type `Shape` defined as `type shape = Triangle of float | Square of float` and a function `area` which takes a `Shape` (by the existing definition), and handles the `Triangle` and `Square` cases separately. Now we want to add a new type of `Shape`, namely a `Hexagon`, and modify `area` appropriately to handle it. To do so, we would like to make a new method `new_area` which takes type: `Shape | Hexagon`, and has two cases: if the input to `new_area` is a `Shape` by the old definition, then we just call the old method on the input; otherwise, we handle the `Hexagon` case separately. This sort of wrapper framework is also extremely analogous to the visitor framework in object-oriented programming.

Type classes essentially formalize this idea of extendible types. More specifically, in OCaml this is done by defining `Shape` as a polymorphic variant, with `type shape = [‘Triangle of float | ‘Square of float]`. Any method which takes in a `Shape` as an argument technically has the type `[> ‘Triangle of float | ‘Square of float]`, which means a type larger than that of our pre-defined `Shape`. Thus, we are able to retroactively add classes and define new functions to support them, which take in this “larger type” as input. For example, we could write `let new_area s = match s with shape as x -> area x | ‘Hexagon len -> 6.0 *. len`.

Qualitatively, type classes are very lightweight, and essentially do a minimal amount of work, with the polymorphic variant handling all of the typing interpretation. However, there are a few difficulties with this solution. As with all variations of the “wrapper” framework of solutions, it can become quite difficult to carry around a new copy of every

function every time we add a class. Furthermore, type safety is somewhat sacrificed, and the inelegant workaround to this consists of annotating every function declaration with types, which can get complicated. A more concise compromise in OCaml is the extensible variant type, where we can formally modify the type `Shape` to include the `Hexagon` case, with `type shape = ..` and `type shape += Hexagon of float`.

Quantitatively, we observed a slight performance hit in our simple example, but we think the performance hit due to overhead will likely be larger the more class extensions there are that need to be supported. We compare an implementation using polymorphic variants with a baseline, which simply declares all the classes and functions in one go and does not support retroactive modification. We attribute the overhead to having to follow different pointers to old functions in the new function definitions.

	Polymorphic Variants	Baseline
Runtime (seconds)	0.977	0.861

Fig. 3. We show the average runtime, in seconds, of our shapes benchmark for both the baseline and polymorphic variants implementations. Times are averaged over 20 trials in order to reduce variance.

5 MULTIMETHODS

Up to this point, solutions we have discussed have followed a fairly similar skeleton: they roughly consist of creating new functions which case on input to call the existing function, or handle new cases, in order to support adding new object classes. This begs the question: what if we relax the problem setting by allowing functions to retroactively be modified by cases? In particular, it would be most convenient if we were able to define some function, say `defmulti area class`, where `class` can be cased on retroactively. In a functional programming language, this maintains the feature that adding new functions is simple, but furthermore it lends itself to the easy adding of classes.

The case study we consider here is multimethods in Clojure which are a simple feature of the language. Here, we follow a solution outlined in [Bendersky 2016]. The primary feature which is exploited here is the fact that we can define "open methods", in other words first class methods which are allowed to act on existing and newly-defined types.

To give a simple example of how straightforward this solution is in practice, we describe how we implemented `area` as a multimethod over shapes. We define the method via `(defmulti area class)` as described above, and define cases on existing classes to the method via `(defmethod area Square [s] (* (:length s) (:length s)))`. If we retroactively define a new class, say `(defrecord Hexagon [length])`, we are able to modify the casing with `(defmethod area Hexagon [h] (* 2.598 (* (:length h) (:length h))))`.

Qualitatively, this is definitely the simplest solution to the expression problem we encountered – implementing it requires essentially no additional work other than declaring that the method in question is a multimethod! This is likely a situation that the makers of Clojure considered in defining the expressibility of the language, and is an advantage to the way it was designed.

Quantitatively, the implementation via multimethods vs. a simple baseline (writing separate functions and hardcoding cases to existing classes) had a slight performance drop, but it is not extremely significant, and is likely due to a slight overhead due to the implicit casing of the method definition vs. the explicit casing defined within the method in the baseline.

	Multimethods	Baseline
Runtime (seconds)	3.935	3.680

Fig. 4. We show the average runtime, in seconds, of our shapes benchmark for both the baseline and multimethods implementations. Times are averaged over 20 trials in order to reduce variance.

6 CONCLUSIONS AND DISCUSSION

We analyzed existing solutions to the expressivity problem, which arises in both functional and object oriented programming languages. We implement and benchmark four solutions to the expressivity problem: object algebras, the visitor framework, type classes, and multimethods. Each of these methods cleanly solve the expressivity problem; however, they all present their own tradeoffs too in aspects such as runtime, ease of use, and type safety. We provide a detailed analysis of the tradeoffs involved for each of these solutions to the expressivity problem in our paper.

We would like to note that all of these solutions to the expressivity problem are variants of the same core idea. Namely, the idea is that to extend functional programming to allow the addition of types, we can extend our functions so that they behave the original way when given types that were already defined and also accommodate the newly added types. Object oriented languages implement this idea by essentially "inverting" to become more functional and then extending the classes which define this functional behavior. Functional programming languages implement this idea in a more direct manner.

To close, we note that all of these solutions work using the functional viewpoint of programming languages - is there a corresponding object oriented solution? This natural question arises from the realization that the first 3 solutions discussed in this paper all follow a very similar skeleton, where if we want to retroactively modify some function `f_old` to case on some class `c_new`, it suffices to define a "wrapper function" `f_new` which has two branches: if the input is of type `c_new`, define the case within the function body of `f_new`; otherwise, simply call `f_old` with the same input. For functional programming languages, this is a natural solution, but its relationship to the visitor framework feels unnatural, in the sense that it is an extension to functions despite the visitor framework working on objected-oriented languages.

To reconcile this inconsistency, we propose the following alternative solution for object-oriented languages, which can be viewed as dual to the solutions discussed in this paper. Suppose we are in an object-oriented framework and we have some class `c_old` which we wish to retroactively modify to support some new function `f_new`. Then, we posit that it suffices to define a new class `c_new` which contains an instance of `c_old` as a field, and also a definition for `f_new` applied in the desired way. In this way, it also supports all previously defined methods without reusing existing code, by simply passing them the computation to the corresponding instance of `c_old`. While it is a simple reversal of several existing solutions, we did not encounter this type of solution in our literature review, and think it may be independently interesting. We hypothesize that it is possible that more overhead is incurred while "layering classes" instead of "layering methods", but it would be interesting to evaluate this type of solution systematically in future work.

Equal work was performed by both project members.

ACKNOWLEDGMENTS

The authors would like to thank Varun and Will for being homies.

REFERENCES

Eli Bendersky. 2016. The Expression Problem and Its Solutions. https://eli.thegreenplace.net/2016/the-expression-problem-and-its-solutions/. (2016).

Antoine Kalmbach. 2016. The expression problem as a litmus test. http://ane.github.io/2016/01/08/the-expression-problem-as-a-litmus-test.html/. (2016).

Shriram Krishnamurthi, Matthias Felleisen, and Daniel P Friedman. 1998. Synthesizing object-oriented and functional design to promote re-use. In *European Conference on Object-Oriented Programming*. Springer, 91–113.

Bruno C d S Oliveira and William R Cook. 2012. Extensibility for the Masses. In *European Conference on Object-Oriented Programming*. Springer, 2–27.