Christopher Wood

Reflection Essay #5

The *expression problem* describes the programming language design issue of being able to define a program (or subset of the program) as a datatype that has a set of cases and associated processes (or functions) that operate on the datatype [2]. The datatype cases and supported processes should both be extensible while maintaining the strong static type safety of the datatype (i.e. processors are only allowed to use the datatype in the way described by their corresponding datatype variant/case). Furthermore, this extension should be supported after compilation in such a way that it does not affect any existing compiled code. As stated by Wadler, the ability for a language to solve the expression problem is a "salient indicator of its capacity for expression" [1].

Based on this description, we can see that this type of problem arises in situations where users may need to extend the operations applicable to built-in and user-defined datatypes. One example where this might occur is in the design and implementation of a graphics editing software tool that uses existing code from two separate graphics libraries. Although I am not speaking from experience, I would imagine there are cases where a developer may want to define new shapes with corresponding transformation functions in order to create different images within the tool. In order to accomplish this task, the developer must add new processors for the types supplied by the two graphical libraries without changing how they are implemented (there may be other code in the graphics library that is dependent their implementation). As an example, the developer may want to use a square type from the first library (represented as a collection of points in a 2D plane) and a triangle type from the second library (represented as a collection of vectors in a 3D plane) to draw a "house" shape. The developer cannot change the triangle code to project the 3D image onto a 2D plane to draw this house because its internal implementation may have many other dependencies. The developer must therefore come up with a way to define the new house shape by extending the underlying types without recompilation, all while ensuring type-safety when using them. Thus, this is a clear example of the expression problem, and there are undoubtedly many more examples where this problem comes into play in practical software development projects.

The utility of studying this problem depends on the perspective in which we view programming languages. If we consider a programming language to be strictly a form of expressing computations, then I would argue that the ability for a language to solve the expression problem becomes a very important indicator of its ability to fulfill this goal. Programmers tend to think about problems in very abstract and tangible ways, and so it would be ideal to not constrain them to built-in types defined at compile time. We naturally attempt to solve problems by reusing solutions to smaller problems. Thus, software solutions may be (and usually are) composed of expressions that solve sub-problems (developers do not like to "reinvent the wheel"). Granted, the developer could always revisit the software implementation and add new datatype variants or supported operations as needed. However, if this can be achieved at runtime with minimal computational overhead for resolving types and ensuring type safety, then we essentially lift the developer's constraint to pre-defined datatypes and operations to solve specific problems. Consequently, the language promotes more natural problem solving through free-form expression, which might make it a "better" language to use in the long run.

Finally, it is interesting to note the *expression problem* solution differences in both the imperative and declarative programming language domains. Wadler's solution is beautifully crafted in the imperative object-oriented paradigm. His application of the Visitor pattern, combined with generics, made it easy to add new cases to the datatype, but the addition of operations is somewhat convoluted. This seems to contradict the solutions presented in the declarative functional domain (such as those implemented in Scala), which tend to provide easy ways to extend the processors for a datatype but make it difficult to include new cases. This key difference seems to provide further support for the fundamental differences between these two paradigms. Although both techniques present viable solutions to the problem, they both have their own implementation benefits, as is the usual case between imperative and declarative languages.

While my primary field of interest is not programming language design, I understand the importance and merit of "programming-languages problems" similar to the *expression problem*. Software engineers are often trained to solve problems within the context of a set of tools, but with novel solutions to these fundamental language design problems we can train future generations of engineers to think in terms of the problem at hand, and then simply find the best tool that fits the job.

References

[1] P. Wadler. The expression problem. *java-genericity Mailing List*, 1998.

[2] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. *URL http://scala. epfl. ch/docu/related. html*, 2004.