Christopher Wood
Reflective Essay #3

In recent years there has been increasing evidence of the integration of programming language paradigms. Perhaps the most significant signs of this merging seen in industry are the presence of lambda operators in highly imperative languages like C# and Python. Such features simplify many tasks in an imperative language, such as searching and filtering data structures. However, while these efforts are the result of merging two separate domains, there has also been similar work done combining multiple declarative paradigms together. Functional logic programming, a dichotomy of separate and seemingly conflicting paradigms, has been shown to be quite useful for a variety of reasons. We present a comparative analysis of the benefits and drawbacks of this new multi-language paradigm in the context of Curry, the most prominent functional logic programming language available.

Perhaps the greatest benefit of multi-language paradigms is the ability to selectively apply helpful programming techniques from any supported language. For example, Curry gives programmers the ability to make use of nondeterminism with free variables and multiple rule declarations, while at the same time supporting functional composition to aid in algorithmic abstractions. The programmer is then left to make use of both of these features at their leisure, depending on the specific problem they may be trying to solve. Using them together can also prove to be very useful as well. Antoy and Hanus [1] present an example of encoding a regular expression matcher in Curry that uses the nondeterminism through multiple rules and functional patterns to determine if a given input string matches an ended regular expression. Fortunately, Curry's demand-driven search strategy limits the computational overhead of the incorporating free variables into function parameters. In addition, the programmer can also modify the rules of the regular expression to place further constraints on the problem (such a technique is referred to as narrowing through "constrained construction"). Put another way, the benefit of having these different language features at your disposal is an attempt to make the language more declarative.

In the context of Curry, another major benefit of its functional and logic juxtaposition is that is enables programmers to more effectively prototype their software solutions to particular problems. If a solution or an efficient algorithm is not known, the nondeterministic nature of Curry, combined with the functional representation of the problem, can be used to explore the steps that are used to find a solution. This might lead to additional insight about the problem at hand and also inspire programmers to devise creative algorithms to generate correct solutions. Speaking in generalities, this is effectively a benefit gained from the ability to encode problem solutions in more abstract descriptions.

Unfortunately, as with any level of software abstraction, there is a price to be paid for the added convenience factors that make software development easier. In Curry, the search strategy employed to intelligently search the state space of a free variable is motivated by demand-driven evaluations, constrained construction, and the use of defitional trees to structure re-write rules. Even with these improvements, the search space of a free variable is still determined by the rule and equation constraints imposed by the programmer. If these are not very well defined and the programmer wants to explore the search space for a solution, then the search strategy cannot optimize its behavior to yield acceptable performance. Thus, we see that an trade-off between performance and easy development arise in such situations.

Furthermore, multi-language abstractions may also introduce conflicts with one of the languages, thus yielding negative effects when programs are run. Although the search strategies and functional composition of Curry both promote parallelism, it is not difficult to imagine cases where two conflicting goals may be sought after. A simple example, consider the attributes of design flexibility and code readability. While multi-paradigm languages may offer more flexibility in design and implementation decisions (simply because they lend more control to the programmer), the additional syntax and program semantics that come with this merge of two language paradigms can burden the programmer and future code maintainers. From my experience in the industry, my usage of lambda expressions in C# code seemed foreign to my coworkers, often to the point where they would not pass my code through inspection unless I used the traditional imperative analogs.

Overall, we see that there are indeed many benefits and drawbacks to multi-language paradigms. Antoy and Hanus have shown us many examples with their discussion of Curry. However, the biggest implication from their article is that it is extremely important to consider all of the implications of a multi-language paradigm before readily adopting it for full-time use. There are many theoretical and practical advantages and disadvantages to such paradigm, and as with any programming problem, the ideal language is best chosen by examining its usefulness in the context of a specific task at hand. There is no “one size fits all” language that any merging of paradigms can solve.

References

[1] Sergio Antoy and Michael Hanus. Functional logic programming. *Commun. ACM*, 53(4):74–85, April 2010.