

# The Ramifications of Programming Language Design on Verifiability

Derek Bronish

**Abstract**—The RESOLVE programming discipline consists of a specification language used to model desired program behavior, and a programming language in which to write implementations. Currently, RESOLVE programs are written in an imperative style for which a sound and relatively complete proof system has been defined. We propose an investigation in to a “functional” programming language, and an examination of its potential benefits both for program design and verification.

**Index Terms**—Verification, functional programming, programming languages

## I. INTRODUCTION

At a theoretical level, all non-trivial programming languages are capable of computing the same things—but of course not all languages are created equal. Matters of convenience, conciseness, clarity, and subjective personal aesthetics all have substantial impact on a language’s prospects for success. An ideal language is one that can be all things to all programmers, catering to their needs and preferences with ease. From the point of view of verification, the key challenge in designing such a language is that all of its variants must be supported by a rigorous mathematical foundation and clearly-defined semantics.

Perhaps the most substantial upshot of the RESOLVE paradigm in this regard is that its theoretical underpinnings—the specification language and mathematical theory libraries—are entirely programming language-agnostic. The primary reason why the RESOLVE programming language is imperative is that there already exists a well-defined semantics and a sound and relatively-complete proof system for establishing correspondence between such programs and RESOLVE specifications [2][4]. It seems that the mathematical core of RESOLVE could underpin countless new programming languages without any changes being necessary, so long as each language’s definition included a semantics and—if verifiability were desired—a proof system.

## II. DISCUSSION

As a first excursion in to this new territory of “branching” novel programming languages from RESOLVE’s mathematical trunk, a “functional” programming language seems like a good candidate. We write “functional” in quotes because a substantial body of RESOLVE-related research (*e.g.*, [5]) makes a compelling case for relational semantics. Indeed, there appear to be serious theoretical roadblocks to the verifiability of a purely functional language in the presence of relational

semantics and abstract data types [1]. Rather than attempting to design a truly pure language, we intend only to explore the pragmatic benefits of a functional *style* for both the code-writer and the code-prover.

Despite the fact that it has been programmed in a procedural style for its entire history, RESOLVE does have features that make the transition to a functional paradigm manageable. The foremost among these is its “clean” semantics, meaning that the execution of a statement impacts only the entities mentioned in that statement. In the functional world, this means that evaluation has no side-effects: a property that eases program proof considerably.

If the development of a Lisp-style functional RESOLVE language were to be undertaken, one of RESOLVE’s best-developed theories (strings) could be used to model the language’s main primitive: lists. An interesting research question is: “how much Lisp-style computation can be specified in the fragment of string theory for which we currently have a decision procedure?” This decision procedure, developed by Harvey Friedman, has been implemented by Bruce Adcock and the author in a tool called SplitDecision. SplitDecision takes as input “verification conditions (VCs)” —logical formulae whose syntax is entirely divorced from that of the programming language. This means that, given a proof system for generating VCs from code and specs, a functional RESOLVE programming language could use the existing program proof tools entirely as-is.

Interestingly, the fact that the math/spec world remains unchanged across RESOLVE programming language variations does *not* necessarily mean that each programming language is equally powerful in terms of what algorithms it can verify. It may be the case that a RESOLVE language designed from a functional mindset would demand a proof system (a method of VC generation) so different from the existing ones that formerly unprovable algorithms become provable. This remains to be seen, but more diversity in the styles and sources of VCs can only help the concurrent research program of investigating automated proof methods.

A final, less abstract reason for pursuing this track is that all existing efforts to verify functional programming languages have differed greatly from the RESOLVE approach. Some very popular systems such as Nqthm/ACL2 [3] are so disparate in in their fundamental views on what verification is and how it should be done that the introduction of a RESOLVE-style system could foster mutually beneficial dialogue between two previously-disconnected research communities.

D. Bronish is with the Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, 43210 USA e-mail: bronish@cse.ohio-state.edu.

### III. CONCLUSION

A programming language designer's work is never finished. There will never be "one language to rule them all." Part of RESOLVE's brilliance is its separation of the mathematical necessities of computing from the syntax in which programs are actually expressed. As yet, this separation has not been sufficiently leveraged to create unique, powerful, beautiful, and varied languages. This position paper advocates the exploration of one such new language, and surely others could follow.

### REFERENCES

- [1] Bronish, D., Kirschenbaum, J., Adcock, B., and Weide, B.W., *On Soundness of Verification for Software with Functional Semantics and Abstract Data Types*, Technical Report #26, Dept. of Computer Science and Engineering, The Ohio State University, Columbus, OH, 2008.
- [2] Heym, W.D., *Computer Program Verification: Improvements for Human Reasoning*, Ph.D. dissertation, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH, 1995.
- [3] Kaufmann, M., Manolios, P., Moore, J.S., *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers, June, 2000.
- [4] Krone, J. *The Role of Verification in Software Reusability*, Ph.D. thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH, 1988.
- [5] Sitaraman, M., Weide, B.W., and Ogden, W.F., *On the Practical Need for Abstraction Relations to Verify Abstract Data Type Representations*, IEEE Transactions on Software Engineering 23, 3 (March 1997), 157-170.