Applying the Lessons of RESOLVE to a Functional Programming Language

Derek Bronish

***Abstract*—No existing verification system for a functional programming language offers a sufficiently powerful combination of abstraction and modularity. The theoretical innovations and overall programming discipline of RESOLVE can be leveraged to ameliorate this situation and accommodate to a new, verifiable functional language.**

I. Motivation

ONE fundamental, yet insufficiently-investigated question in the study of software verification is: "how does the nature of a programming language impact its prospects for verifiability?" In some situations the answer is clear, *e.g.,* a language with reference semantics is harder to reason about than one with value semantics, but in general the ramifications of programming language design on verifiability are subtle, and demand further research.

Putting one's self in the shoes of a language designer who seeks to facilitate both verifiability and elegant code, a pure functional language seems like a natural choice. The reasons are plentiful; we list just a few:

- Operations have no side-effects, so the order of sub-expression evaluation need not be accounted for in the verification conditions.
- The language primitives (*e.g.,* lambdas, s-expressions) are more amenable to mathematical formalization than those typical in imperative languages (*e.g.,* arrays, pointers, bounded integers).
- Writing non-trivial code in a functional language, more so than in an imperative one, forces the programmer to construct a hierarchical decomposition of the problem in to pieces. Modular design is not easily avoided.

If these reasons are at all persuasive, the reader should then wonder if a verifiable pure functional language already exists. Surprisingly, the answer to this question is a near-resounding "no."

Far and away the most popular functional language verification system is ACL2 [1], a prover for a slightly-modified subset of Common Lisp. Although it is richly developed and widely adopted for verification case studies[1], ACL2 has substantial deficiencies, particularly in its accommodations (or lack thereof) for new user-defined data types. Indeed, in [2] the author flatly states: "ACL2 does not support the introduction of new data types."

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

[1]The interested reader can find many examples at http://userweb.cs.utexas.edu/users/moore/publications/acl2-papers.html

This state of affairs does not appear to be due to any deep theoretical impediment. Rather, the developers of ACL2 simply restricted their language in a way that was expedient for their purposes. The result is a system of substantial power and robustness, but also one which forces program designers to take a low level view of their design, beset with inessential complexities.

Haskell [3], on the other hand is a pure functional language with ample accommodations for user-defined data types. However, no comprehensive effort to verify the language has yet been undertaken. Implementations of the language build in some static analysis in the form of a type inference/type checking system, and extending this functionality with verification seems like a golden opportunity for applying the lessons of RESOLVE to a functional language.

We should be careful at this point not to give the impression that leveraging RESOLVE verification for Haskell is simply a matter of creating a new verification condition generator to account for the language's different syntax. Indeed, substantial research results will be necessary before Haskell verification becomes possible.

II. A Soundness Issue

In [4] we expressed concern that a language with functional semantics in which abstract data types and relational specifications may be written is vulnerable to an unsoundness in the verification of ADTs. Overcoming this vulnerability may require both syntactic modifications (*i.e.,* "flagging" operations whose specifications license relational behavior) and a proof strategy that differs from the traditional inductive approach of [5]. An initial idea along these lines is that one of Haskell's most mysterious and powerful features, monads, might be used to "carry" the "envelope" of a relationally-specified object's possible values along with the particular value that the language's functional semantics will result in. Currently, a real treatment of this issue remains future work.

III. Verifying "Algebraic" Data Types"

Haskell affords programmers the opportunity to define new data types in a "purely algebraic" fashion, meaning that constructors for object values are defined, and then the effects of the type's operations are expressed in terms of these constructors. As an example, a simple `Stack` data type can be defined with constructors for empty, singleton, and "size at least 2" stack values:

```
data Stack t = Empty
             | Singleton t
             | Usual t (Stack t)
```

And then the `Length` function is easily defined by cases depending on the nature of the `Stack`:

```
Length :: Stack t -> Int
Length (Empty) = 0
Length (Singleton _) = 1
Length (Usual x s) = 1 + (Length s)
```

Note that although we are implementing `Stack`, we are not choosing a representation for the data. The language implementation is free to store `Stack` values in whatever manner it likes, so long as it is memoized in a manner that allows the constructor-based pattern matching of the operation definitions to be carried out. Exactly how to verify such implementations is an interesting research question.

Interestingly, although Haskell allows such "purely algebraic" implementations, it does not *require* them. Instead, we as implementers can choose that we want our `Stack` entries to be stored (for example) in a list, and implement the operations accordingly:

```
data Stack t = ListOfEntries [t]

Length :: Stack t -> Int
Length (ListOfEntries l) = length l
```

where `length` is a built-in function for computing the length of Haskell lists.

An obvious first step in research along these lines is to write multiple implementations of some data types in these different styles, and investigate which style tends to be most amenable to verification. Perhaps the "obviousness" of the relevant verification conditions will depend on the style in which a type is implemented.

IV. Conclusion

We have described a research agenda whose goal is to create a language with "pure functional" style that can accommodate sound and relatively complete verification. An initial investigation has revealed Haskell to be a good starting point, and key research issues involving relational specifications and algebraic data types have been identified. We are eager to continue this work, and hope that it results in a flow information that is mutually beneficial between the verification and the functional programming communities.

References

[1] M. Kaufmann, J. S. Moore, and P. Manolios, *Computer-Aided Reasoning: An Approach*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[2] J. S. Moore, "Finite set theory in ACL2," in *Theorem Proving in Higher Order Logics*, ser. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, 2001, vol. 2152.

[3] S. Thompson, *Haskell: The Craft of Functional Programming (Second Edition)*. Addison-Wesley, 1999.

[4] D. Bronish, J. Kirschenbaum, B. Adcock, and B. W. Weide, "On soundness of verification for software with functional semantics and abstract data types," Department of Computer Science and Engineering, The Ohio State University, Tech. Rep. 26, 2008.

[5] C. A. R. Hoare, "Proof of correctness of data representation," *Acta Informatica*, vol. 1, no. 4, pp. 271–281, 1972.