1 Introduction

Standards define interfaces so that useful things can be connected. It didn't much matter whether Fortran used READ and WRITE for I/O (as it does) or GET and PUT (as PL/I does), as long as programs and compilers agree, and that agreement is a standard.

When a standard is defective, useful things made in good faith in the belief that they conform to the standard cannot be connected correctly. Standards being the work of human hands, they always are defective, which is why standards processes include Interpretation Requests, Technical Corrigenda, and periodic revision.

The practical usefulness of a standard may be limited if it omits so much relevant to its application area that there is little benefit from using it, or if it is so large that it is too expensive for people to work to. This paper is only concerned with *defects*, where a standard fails to correctly express a coherent intention.

Some kinds of defect include

- incompleteness: the standard uses terms or interfaces that it neither defines explicitly nor includes by reference from other standards. Example: a standard that calls for fixed-point arithmetic and defines its semantics by reference to LIA, which explicitly has nothing to say about it.

- incoherence: part of the standard fails to be interpretable due to garbled text, formulas, diagrams, and so on. Example: a standard which has not been adequately proof-read, or in which there are errors in a mathematical formula.

- contradiction: it is not possible to work to the standard because it makes conflicting requirements. Example: a standard which specifies the operands of a function in one order in one place and the opposite order in another place.

- inconsistency: the standard is not compatible with other standards that are needed with it. Example: a standard which requires both that timestamps support calendar arithmetic accurate to at least 1 second accuracy into the indefinite future and that timestamps use UTC.

Formal specification techniques *can* help to reduce defects. They do not *necessarily* do so unless they have appropriate tool support. Many people regard such techniques as unreadable by the target audience. Algol 68 was a much simpler language than its formal specification made it appear, and the PL/I standard is little used.

One approach is to use stylised or semi-stylised text. The ECMAScript and Prolog standards are examples of this. Attempto Controlled English shows that it is possible to have a useful notation which reads as English but can also be syntax-checked and to some degree semantically analysed by computer.

Another approach is to use formal methods for *parts* of standards. The use of regular expressions and context-free grammars to specify the syntax of programming language is generally accepted as accessible to the readers of standards.

This article argues that it is possible to use some modern programming languages for formally specifying parts of standards, using fixed-point arithmetic as the topic and Haskell as the programming language.

Using a modern functional language allows these checks:

- definitions are well formed (tokens and syntax)
- definitions are complete (everything that is used is defined, part of type checking)
- definitions are unambiguous (everthing has one definition)
- definitions are used
- definitions are semantically well formed (rich type checking)
- definitions satisfy stated properties (done either by using a testing framework or a verification tool)
- definitions are tested (test coverage)

The Algol 68 and PL/I standards and the formal specification developed for Ada 83 and the formal specification annex of the Prolog standard each used a notation which the reader had to learn for that standard alone. Using a modern functional language means using a notation which is documented in books and tutorials and has adequate tool support for the purpose.