CMU-CS-81-144

On Intensionality and Referential Transparency

Joseph Bates

6 November 1981

Abstract

In designing a formalism for reasoning about functions, one must decide just what functions *are*. This brief note suggests that however mathematicians stand on the issue, programmers must take the view.that functions are methods of computation, not input/output relations.

This research was supported in part by the National Science Foundation under grant NSF-80-03349. It was performed while the author was visiting CMU-CSD from Cornell University.

Introduction

Programmers expend much effort reasoning. This observation has lead to the development of automated "reasoning assistants", such as Automath [4], Edinburgh LCF [5], and PL/CV2 [2]. To construct such an assistant, one must have an extremely precise description of the valid means of reasoning. Such a description is a formal logic.

Functions are one of the main sorts of objects that programmers manipulate. The problem of formalizing reasoning about functions has been studied by many people: it is of interest to philosophers, practicing mathematicians, programming methodologists, and programmers. Since mathematicians have been reasoning about functions for a long time, logics of programming tend to build directly on (mainstream) mathematical work. While much of that work is extremely relevant to programming, there are significant differences between the two fields. These differences suggest that the treatment of functions in programming logics should differ from that developed in mathematics.

When deciding how to formalize functions, the essential question is "what *are* functions?". This is an issue of how we speak of functions: how do we build them, how can they be used, and in particular, when are two functions equal[1] ? This last question is often ignored, for the answer is so obvious: two functions are equal just when they produce equal results on equal inputs, i.e., when they are extensionally equal. However, this is a decision, even if an unintentional one, so we might discuss why extensionality is so often chosen.

Why is Extensionality Assumed?

Mathematicians are seldom concerned with the computational complexity of algorithms. Nor are they interested in the problems of building large, structurally complex, algorithms. Therefore, to simplify life, they usually ignore details of how algorithms work. Their natural notion of function is one of input/output relation, a function is a set of ordered pairs. This view admits very nice formalization - functions are tractable mathematical objects with elegant properties.

Because the mathematician's view provides such a pleasant foundation, and because most computer scientists have been trained with that background, it is natural that extensionality is adopted as the proper basis for a formal treatment of programs. Further, notions of referential transparency, modularization, and information hiding seem to lead towards extensionality. Thus, extensionality is not only familiar and "intuitive", but consideration of real programming issues appears to require it. Nonetheless, let us probe a bit further.

---

[1] We will use this phrasing even though the notion of "two objects" seems to imply that the objects are distinguishable, while equality implies that they are identical. We could be more precise by speaking of two distinct names that denote the same object.

A Deeper Look at the Assumptions

First off, arguments that a certain viewpoint is intuitive or natural are usually of dubious value. Such claims often mean "I've done it this way for a long time and I'm comfortable with it". The superiority of assembly language over FORTRAN and the advantages of unconstrained GOTOs over structured program composition, along with a multitude of other untenable positions, have been supported on these grounds. Further, one's intuition is fluid, gradually adapting to experience and belief. We thus dismiss the argument from familiarity, for it must give way to considerations of substance.

Programmers do not live in as idyllic a world as mathematicians. Computational reasoning covers a broad spectrum of concerns: does a function compute the desired results, is it efficient, is it cleanly structured? These questions are dealt with constantly and a theory of programming must admit their expression and resolution. Clearly though, they are not all properties of the extensions of functions. Instead they are properties of how functions are constructed. Thus, a mature logic of programs must support intensional reasoning.

What about the problem of tractability? As mentioned, the conventional view of functions leads to elegant formalizations. Can as intensional approach be feasible?

At the moment programs are almost laughable as mathematical objects. Though we have several elegant ways of explaining their meanings, the relationships between programs are weak and general laws usually aren't. A manifestation of this is seen in work on gathering programming knowledge. The Irvine Program Transformation Catalog [8] and the Programmer's Apprentice Library [6] each contain hundreds of generally useful transformations. This is not a failure of these efforts. The domain of programs they are trying to capture is inherently messy.

Despite (or because of) the current state of affairs, there is hope that we may be able to develop a better view of programs. Recent work on functional programming is explicitly driven by the desire to have a tractable mathematical domain of programs [1, 7]. For the same reason, combinators have found application in the PL/CV3 programming logic [3]. There are serious problems with these approaches, but our response certainly should not be to discount the entire area. Rather, we should pursue with vigor the construction of a beautiful domain of programs.

The final issue is that of referential transparency. Even if we have a clean formalization of programs, isn't there something nasty about extensionally equal programs that are not interchangable?

No. Intensionality is *not* nasty. We should expect interchangability of equal functions, and for some

applications, equality may be taken as extensional. Yet, much of our discipline is founded on the differences between extensionally equal functions. If extensionally equal functions were indistinguishable, we would lose our ability to discuss code optimization, asymptotic complexity, and elegant programming style, among other fascinating and useful topics. Thus, extensionality is simply inappropriate - it is too coarse an equality for much of our work.

Extensionality may also be too fine. One can argue that good programs are modularized, in that interfaces are specified and modules are constructed to satisfy those specifications. The important property is that any module be replaceable by another module meeting the relevant specifications. This notion of interchangability is far broader than extensional equality, for the specifications characterize whatever equivalence is suited to the problem. Thus, extensionality is inappropriate here, as well.

We really want a logic with two notions. First, when are two functions identical? As argued, this is not an absolute concept. We choose it as is convenient: probably ignoring comments and spacing, perhaps disposing of variables, but retaining as much structure as is mathematically tractable. Second, when does a function meet a specification? This entails the presence of specifications, ways for writing them, reasoning about them, and using them in building functions.

Extensionality is a misguided attempt to meet these two needs simultaneously. It is a useful notion, but it is not fundamental. Instead, along with other equivalences, it should be defined using the logic and the appropriate theorems derived. If the logic has any expressive power, it should at least get us this far.

Philosophical Considerations

There is a somewhat deeper issue here. One may grant that a programming logic should allow us to reason about programs, but assert that it should be built on extensional foundations. That is, one may believe the mathematician's idea of function is primary. This is a subtle matter, but several arguments can be mustered for intensional foundations.

The pragmatic argument is that since we want to reason with intensionality and specification refinement, let us attack them directly and formalize the logic we want to use. If someone feels they can explain the logic in a prior extensional framework, fine, but let us leave such explanations to metatheory.

The second argument is less concrete. If the purpose of the logic is to express our most basic concepts of computation, we must try to understand what is "real". Taking an extensional view requires the notion of infinite set. Are infinite sets real? Do functionals operate on infinite sets? The intensional view requires one only to accept symbols and rules for manipulating them. These are finite things, that we can speak of, if we

choose, as reprcsenting infinite sets.

It is not apparent that there is any sense to the notion of infinite objects. The universe doesn't appear to admit such things, so we have never experienced them. When we write descriptions, such as 0,1,2,...,ω, it is the manipulation of the descriptions that we understand. It may appear that the things we imagine descriptions denoting are real, but this is only because we become so fully immersed in our language that it disappears from our consciousness. Once this occurs, we seem to be left facing the imagined things themselves. This is a convenient fantasy, but a fantasy nonetheless.

Our choice of descriptions and manipulations certainly is influenced by our attempts to imagine actual infinite objects, but as philosophers have found, our intuition about those objects is often faulty. If we are to understand and formulate the underlying concepts of computation, we must stand on solid ground. Accepting language as the basis is hardly an impoverished view. Individuals can still become sufficiently immersed that symbol manipulation becomes subconscious. One can still have a theory in which N->N is not denumerable. The advantage is that the theory is built on concrete and transparent principles.

This last (vague) discussion is intended to show that the classical foundations of mathematics are subject to question. However, this is independent of the pragmatic argument, for it should be apparent that however one makes sense of an intensional logic, such a logic is appropriate and necessary to adequately formalize computational reasoning.

References

[1] Backus,J.
Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs.
CACM (21,8), August, 1978.

[2] Constable,R.L., and M.J. O'Donnell.
A Programming Logic.
Winthrop Publishers, Inc., Cambridge, Mass., 1978.

[3] Constable,R.L., and D. Zlatin.
Lecture Notes in Computer Science: The Type Theory of PL/CV3.
Springer-Verlag, 1981.

[4] de Bruijn,N.C.
A Survey of the AUTOMATH Project.
In Seldin,J.P., and J.R. Hindley (editors), *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism,* . Academic Press, 1980.

[5] Gordon,M., R. Milner, and C. Wadsworth.
Edinburgh LCF.
Technical Report CSR-11-77, University of Edinburgh, Edinburgh, Scotland, September, 1977.

[6] Rich,C. and H. Shrobe.
Initial Report on a LISP Programmer's Apprentice.
IEEE Transactions on Software Engineering SE-4,6, November, 1978.

[7] Sintzoff,M.
Proof-Oriented and Applicative Valuations in Definitions of Algorithms.
In *Functional Programming Languages and Computer Architecture.* ACM, October, 1981.

[8] Standish,T.A., D.C. Harriman, D.F. Kibler, and J.M. Neighbors.
The Irvine Transformation Catalog.
Technical Report, Dept. of Computer and Information Science, U. of California at Irvine, 1976.