The Type Structure of CAT

J. Paul Myers, Jr. and Ronald E. Prather
Trinity University

Computer science educators have long lamented the fact that so many students show a lack of basic understanding of the fundamental mathematical principles underlying the field. Small wonder, when they are taught that "the integers" end at *maxint,* that fractions are to be rounded off to the nearest decimal, and that recursion is something connected with the notion of "pointers." We may think that we are doing an admirable job with our elementary discrete mathematics course; we may be encouraging any number of students to enroll in substantive courses in logic, algebra, and the like. But so long as we persist in the use of programming languages that are counter to the most fundamental mathematical precepts, obscuring their very nature, we cannot hope that the situation will improve. In this paper, we introduce the type structure of a new programming language, CAT, based in the mathematical theory of categories. Among its many novel features, the language offers an exact arithmetic, a blend of applicative and imperative programming methodologies, strong and consistent typing, a functorial semantics, a proof technique, functions as "first class" objects, and recursive structures without pointers. It is felt that a strong pedagogical basis can be achieved by the early introduction to such a language -- provided of course, that the more esoteric categorical properties are kept "at a distance." The CAT language, even as seen at an informal level (as is our treatment here), nevertheless introduces a number of important constructive notions, in particular that of "provable recursiveness," one that seems to have been overlooked in the literature. In the context developed here, it is seen to form the basis for a whole new programming philosophy.

1. Introduction

In the development of the traditional programming language, we more often find that the applications, the machine model of the computing environment, and the habits and attitudes of the common programmer, in some combination, serve to determine the design philosophy and the overall characteristics of the resulting language. Usually these considerations are mutually competitive and we then obtain something of a compromise in design philosophy and in language features. When, at a later stage, we undertake a study of the semantic interpretation of such languages, it is not surprising that all mathematical machinery adequate to the task is found to be cumbersome, unwieldy, and largely unsatisfactory [1].

We believe that the problem is, in part, a question of "domains," and of the necessary "exception handling" when these computational domains are not well chosen. In pure Lisp [2], we have an unusually simple domain structure, all computational objects are of one type, and accordingly, the semantic interpretation of Lisp is quite straightforward. On the other hand, not everyone is completely happy with such a rudimentary programming environment, however elegant the foundation may be.

In a previous paper [3], one of the authors has proposed an equally simple programming language foundation, drawn from the mathematical theory of categories [4,5], yet having the consequence that more conventional programming structures and techniques might result. Here, it is our task to describe the elements of such a language in somewhat more detail, with particular attention to the underlying type structure, i.e., the nature of the domains to be admitted. Since we eventually arrive at the situation where, as they say [6], "functions are first class objects" (themselves members of some admissible domain), we give an almost equal treatment to these functions, alias morphisms of the category, alias provably recursive programs.

2. The Category of Provable Recursiveness

In the earlier paper [3], it was first suggested that the category (here called **RecSet**) consisting of *recursive sets* as "objects" and (total) *recursive functions* amongst them as "morphisms," might serve as the cornerstone to the CAT language development. Certainly if we are given $f : A \to B$ and $g : B \to C$, the ordinary composition $g \circ f$ is again recursive, and the categorical axioms:

$$\begin{array}{ll} \textit{associativity} & h \circ (g \circ f) = (h \circ g) \circ f \\ \textit{identity} & i \circ f = f \quad \text{and} \quad g \circ i = g \end{array}$$

are clearly satisfied, where $i = i_B$ is the *identity* morphism (recursive function) on B. Moreover, the resulting category **RecSet,** a subcategory of **Set,** has a number of useful properties (e.g., the monomorphisms are the injective morphisms, and dually, the epimorphisms coincide with the surjective morphisms, as in **Set**). But in yet another setting [7], the same author provides a convincing argument that **RecSet** finds its best use as a category for the semantic interpretation of CAT, whereas it falls short of the original purpose owing to the fact that it is not a "cartesian closed" category [5], a consequence of the unsolvability of the halting problem [6].

So instead, we are led to introduce as our programming language foundation, the category **ProRec** of *provable recursiveness,* and ultimately (as in [7]), a semantic functor [5]

$$\mathbf{S}: \mathbf{ProRec} \to \mathbf{RecSet}$$

in which the domains of CAT (objects of **ProRec**) and the "provably" recursive functions (morphisms of **ProRec**) are each given a concrete interpretation. Basically, the objects of **ProRec** consist in the smallest class of domains or "types" containing the (six) elementary domains to be presented in the next section, and closed under certain (again, six in number) type constructions, including recursion, as discussed in subsequent sections. And as morphisms, the *provably recursive functions* $f : X \to Y$ among such domains are defined as in the program schemata:

$$\textbf{function } f(x : X) : Y;\ C\ \{P\}$$

where C is a (most likely compound) command and P is a proof that for each x in the domain X, the execution of C yields a value for f in the domain Y. Both the objects and the morphisms of **ProRec** are subject to a notion of "structural equivalence" [9], causing an identification of suitably equivalent objects (resp. morphisms). But in the case of the latter, we hasten to observe that such equivalence is largely only textual, i.e., many different programs will still compute the same function,