1 Introduction

The Reader who knows that polymorphism is not set-theoretic (cf. [16]) is nevertheless implored not to quit reading out of hand. We are aware that the complexity of full polymorphism escapes a set-theoretic description. The situation, however, is not as hopeless with many restricted forms. This includes shallow polymorphism such as in ML. The aim of this paper is to define precisely the set-theoretic interpretation of the ML-like polymorphic system of types. The aim of this Introduction is to convince the Reader that this is worthwhile to try.

1.1 The use of set-theoretic interpretations

One reason for defining semantics of a formal system is to enhance its understanding. A typical semantic function should give a meaning of syntax within a world one knows well beforehand. For instance, a definition of semantics which interpretes computer programs as functions (that transform states) is useful to people with a good understanding of the concept of function.

If an attempt to define semantics within a given world fails one may want to replace this world by a different one; one may even want to construct a special domain of denotations for the particular formal system. But while doing so one may end up with a consistent but unintuitive model with little explaining power.

The authors of this paper, and in fact the MetaSoft group, are convinced that this happens to many formal systems in the computer science.
The simplest example is the untyped \( \lambda \)-calculus. Originally designed to investigate the ways functions apply to objects, it turned out to have no set-theoretic model, i.e. no model in which \( \lambda \)-expressions were interpreted as functions and the operation of application as the genuine application of a function to an argument.

However, there are numerous models of \( \lambda \)-calculus, for instance \( P_\omega \), the powerset of natural numbers, proposed by D.Scott [17]. Unfortunately, all these models are highly unintuitive. For example, to explain the meaning of "\(((\lambda x)z + 1).2\)" in \( P_\omega \) a subset of \( \text{Nat} \) is assigned to "\((\lambda x)z + 1\)" and another subset to "\(2\)", the both subsets are treated by the special "application" operation that has nothing to do with the true function application; finally a subset of \( \text{Nat} \) is yielded as result.

We think it possible to develop the "naive" semantics that respects the intuitive understanding of such basic mathematical concepts as function and function application, set and set belonging, equality, etc. We cannot construct such a model for the full untyped \( \lambda \)-calculus because it involves self-application which is not set-theoretic. But we do believe a user has always in mind its typed version without self-application. We cannot solve any type equations but we are happy with the useful ones that do not require reflexive domains. This "naive" approach has been presented in [4].

In this paper we give a set-theoretic model for polymorphism. We hope the model is both consistent and intuitive. We believe our approach is adequate for discussing polymorphism and also for explaining the concept to uninitiated.

1.2 Discussion of models for polymorphism

We understand (parametric) polymorphism as a dependence of objects and types on types. The concept appears in a number of programming languages in one form or another, for instance in ML (cf. [11]) and in ADA (cf. [1]). The distinction between the explicit polymorphism of ADA and the implicit one of ML is not essential for our considerations: whether type parameters are explicitly present in expressions or inferred automatically using Milner's algorithm from [13] is a syntax issue; we assume they are around whenever we need them to discuss semantics.

When we say that a polymorphic type is a dependence of types on types, i.e. a function

\[ T : \text{Types} \rightarrow \text{Types} \]

we have to be careful to properly define the \( \text{Types} \). In particular, it is important to realize whether polymorphic types, such as \( T \) above, belong themselves to \( \text{Types} \).

In Girard's and Reynold's second order polymorphic \( \lambda \)-calculus [9,15] a polymorphic type may be instantiated by itself. This causes non-existence of set-theoretic model for it (cf. [16]) similarly to the case of the untyped \( \lambda \)-calculus. The construction of a model by Pitts [14] does not falsify this