Typed $\lambda$-calculus with recursive definitions

Sergey V. Kotov
Institute of Mathematics,
Universitetski 4, Novosibirsk 630090, Russia

Abstract

This paper introduces an extension of a typed $\lambda$-calculus with locM recursive definitions, that is, a language combining higher types and recursion. In contrast to the Gödel's calculus of recursive functionals, the reduction presented arranges local definitions to a normal form. Essential properties of the reduction are soundness with respect to the outlined categorical semantics, confluence, and strong normalization. Suitability of the language to be a basis for higher order specification languages is discussed.

1 Introduction

There are at least two points of view on the role of $\lambda$-calculi in logic and computer science. First, it is a paradigm of sequential execution and computation; and, second, (typed) $\lambda$-calculi can serve as a skeleton of higher-order formalisms. A number of extensions of $\lambda$-calculi have been developed. Gödel [8] proposed a language which can be viewed as a $\lambda$-calculus with recursion over natural numbers (e.g., see [20]). The meaning of a recursive term is given by the reduction rule which calculates recursive expressions over numbers, that is, the language embodies the first point of view. We consider a typed $\lambda$-calculus with recursive definitions developed with the second point of view in mind; recursion is not eliminated by reduction and terms are reduced to their normal forms rather than executed. Here we follow the reduction second-order calculus of Moschovakis [15] designed to be a foundation for recursive algorithms theory. The normal form represents recursion uniformly, and we expect this form to be useful for reasoning about higher order recursive expressions.

Thus, we refer to our formalism as a kernel of higher-order specification languages. Such languages usually include a parametrization mechanism and a number of specification-building operations. One of the approaches to parametrization employs $\lambda$-style (see [11, 21]). In this case, the operations form a (many-sorted) algebra; and a $\lambda$-formalism is placed on the top. Our calculus has two features:

- the parametrization mechanism is not on the top, but it is developed simultaneously with the specification-building operations discipline; thus, the question "can parametrized specifications be structured?" is solved automatically;
- local, possibly recursive, definitions are allowed.
We hope the uniform representation of recursive definitions developed in this paper will simplify proofs of properties of specification languages, and, in particular, the proof of correct implementations.

Yu.L. Ershov [4] introduced the language of so-called Σ-expressions. All basic features of our language appear there in a slightly different form. “Flat specifications” are formulae; operations are logical connectives and quantifiers; recursion and λ-abstraction are allowed. However, questions of a reduction calculus and specification language applications addressed in this note were not studied.

The note is organized as follows. Section 2 provides syntax; in Section 3 we outline a denotational semantics of our language in terms of cartesian closed categories (ccc). The semantics follows the well-known correspondence between ccc and typed λ-calculi. Interpretation of recursive definitions is based on the work of Barr [2]. Section 4 introduces a sound, confluent, and strongly normalizing reduction. This makes possible to derive a normal form with at most one occurrence of a recursion operator. An application of our formalism as a kernel of a parametrized structured specification language is discussed in Section 5.

2 Syntax

Let $S$ be a given set of sorts.

Definition 2.1 Type expressions (types) over $(s \in) S$ are defined by the grammar

$$t ::= 1 \mid s \mid (t \times t) \mid (t \rightarrow t).$$

$T$ stands for the set of all types.

We abbreviate $(t_1 \times (t_2 \times \ldots \times t_n) \ldots))$ as $(t_1 \times t_2 \times \ldots \times t_n)$; if $n = 1$ then $(t_1)$ is simply $t_1$; for $n = 0$ we obtain 1. Types of the form $((s_1 \times \ldots \times s_n) \rightarrow s_0)$ where $s_0, s_1, \ldots, s_n \in S$ are called flat.

Suppose we are given

- an infinite set $V$ of variables indexed with types;
- a set $F$ of functional symbols indexed with flat types.

Our language includes surjective pairing, projections, λ-abstraction, application, recursive expressions, and symbols from $F$ as constants.

Definition 2.2 The set of terms $E$ indexed with types is defined inductively as follows:

1. $V \cup F \subseteq E$;
2. $e_1 : t_1, e_2 : t_2 \in E \implies (e_1, e_2) : (t_1 \times t_2) \in E$;
3. $e : (t_1 \times t_2) \in E \implies \pi_1(e) : t_1, \pi_2(e) : t_2 \in E$;
4. $e : t_1 \in E, x : t_2 \in V \implies \lambda x. e : (t_2 \rightarrow t_1) \in E$;