Type reconstruction in $F_\omega$ is undecidable *

Paweł Urzyczyn
Institute of Informatics, University of Warsaw
ul. Banacha 2, 02-097 Warszawa, Poland
urzy@mimuw.edu.pl

Abstract

We investigate the Girard's calculus $F_\omega$ as a "Curry style" type assignment system for pure lambda terms. We prove that the type-reconstruction problem for $F_\omega$ is undecidable (even with quantification restricted to constructor variables of rank 1). In addition, we show an example of a strongly normalizable pure lambda term that is untypable in $F_\omega$.

1 Introduction

The system $F_\omega$ was introduced by J.Y. Girard in his PhD Thesis (an English reference is [8]) as a tool to prove properties of higher-order propositional logics. The system is an extension of the second order polymorphic lambda calculus, known also as "system F". The extension allows for (quantification over) not only type variables, but also variables for functions from types to types, functions from functions to functions and so on, i.e., involves an infinite hierarchy of type constructors classified according to their ranks in a similar way that types are divided into ranks in the finitely typed lambda calculus. A general exposition of the properties of $F_\omega$ can be found in [4]. See also [3], [1], [7] and [21], the latter recommended for instructive examples.

Girard's systems $F$ and $F_\omega$ are "Church style" calculi (in the terminology of [1]): types are parts of expressions, and there is nothing like "untypable term", since an untypable term is just not a term at all. There is however a direct connection between this presentation and the untyped lambda calculus: by a "type erasing" procedure we obtain a "Curry style" system which assigns types to pure lambda terms. This translation between typed and untyped terms creates the type reconstruction problem: for a given pure lambda term, determine whether it can be assigned a type or not. To be precise, terms with free variables may be typable or not, depending on types assigned to their free variables, by a type environment. We write $E \vdash M : \tau$ for "$M$ has type $\tau$ in the environment $E$". Thus, our problem has to be either restricted to closed terms or to be (equivalently) stated as follows:

Given $M$, does there exist $E$ and $\tau$ such that $E \vdash M : \tau$?

*This work is partly supported by NSF grant CCR-9113196 and by Polish KBN Grant 2 1192 91 01
The first paper we are aware of addressing the type reconstruction problem for system F is [17]. The decidability of this problem is still an open question, despite many attempts to solve it. There are several partial solutions, applying either to fragments of system F, or to certain variations of the system, see [2], [11], [6], [7]. An exponential lower bound for time complexity of type reconstruction for F has been obtained by Henglein [9], see also [10] for a complete exposition. This lower bound is quite an involved refinement of the following simple observation: there are terms of length $O(n)$, typable in F, but such that their shortest types are of depth exponential in n (when drawn as binary trees). Consider for example terms of the form $2(2\ldots 2(K)\ldots)$, where 2 is the Church numeral $\lambda x. f(fx)$, and $K = \lambda xy. x$. The method of proof is to replace $K$ with a tricky representation of one step of a Turing Machine, so that the whole term can represent an exponentially long computation. This TM simulation is then put in such a context that a failing computation forces untypability.

A related issue is type reconstruction for the language ML and its modifications, see e.g. [12], [13], [14], [18] (the latter two merged into [15] for a final presentation). For instance, the above mentioned result of [9] uses a technique of [15], developed initially for ML, and the semi-unification problem of [13] is used as a tool in [7].

The type reconstruction problem for $F_\omega$ has a shorter history. A modification of $F_\omega$, similar to that of [2], was shown in [20] to have undecidable type reconstruction. Another variation of the original problem: the "conditional" type reconstruction is also undecidable by a result of [7]. Finally, a nonelementary lower bound has been obtained by Henglein and Mairson [10], using a similar method to that mentioned above. This time, instead of composing 2's as in the previous case, one can consider terms of the form $222\ldots 2$ and apply it to the Turing Machine simulator. A simplified version of this construction is the following exercise: assign types to terms of the form $222\ldots 2K$.

An explanation is in order here: type reconstruction for a system that types all strongly normalizable terms (e.g., involving intersection types) is immediately undecidable ([17]). Thus, one has first to ask whether the set of typable terms is a proper subset of the strongly normalizable terms. For system F, the first example has been given in [5], more examples can be found in [19]. The example of [5] is typable in $F_\omega$, which means that the latter can type strictly more than F. (We conjecture that all the examples of [19] are typable too.) However, we show below (Theorem 3.1) that $F_\omega$ cannot type some strongly normalizable terms. Warning: it is known that the classes of integer functions representable in our systems are all different, and are proper subclasses of the class of recursive functions. However, the fact that a certain term (function representation) cannot be assigned a particular type (integer to integer) does not mean that it cannot be assigned any type at all. In fact, all recursive functions can be represented in the untyped lambda calculus by terms in normal form and all normal forms are typable in F. Thus, results like that of [16] have little to do with our consideration.

The main result of this paper is that type reconstruction in $F_\omega$ is undecidable (Theorem 5.1). The proof is essentially based on the following trick. We construct a term $W$, which is (roughly) of the form:

$$W \equiv \lambda x. \text{if } \text{Final } x \text{ then } \lambda w. \text{nil else } \lambda w. w(\text{Next } x)w \text{ nil}.$$