Information and Computation 248 (2016) 195–214

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

A type assignment for λ-calculus complete both for FPTIME and strong normalization ☆

Erika De Benedetti, Simona Ronchi Della Rocca *

University of Torino, dept. of Computer Science, Corso Svizzera 185, 10149 Torino, Italy

ARTICLE INFO

Article history:
Received 10 December 2013
Available online 6 January 2016

ABSTRACT

One of the aims of Implicit Computational Complexity is the design of programming languages with bounded computational complexity. One of the most promising approaches to this aim is based on the use of lambda-calculus as paradigmatic programming language and the design of type assignment systems for lambda-terms, where types guarantee both the functional correctness and the complexity bound. Along this line, we propose a system of stratified types, inspired by intersection types, where intersection is a non-associative operator. This system is correct and complete for polynomial time when a uniform coding scheme is considered; moreover, all the strongly normalizing terms are typed in it, thus increasing the typing power with respect to the previous proposals, based on Light Logics.

1. Introduction

The importance of controlling (and producing a formal certification of) the resource usage of programs is already recognized by the scientific community. In this general setting, we are interested in the design of programming languages with an intrinsically polynomial computational bound. We are interested in an ML-like approach, so our starting points will be:

- the use of λ-calculus as an abstract paradigm of programming languages;
- the use of types to certify program properties.

In this line, the aim is to design a type assignment system for λ-calculus, where types certificate both the functional correctness and the polynomial bound of terms. There are already two proposals along this line: the systems DLAL (Dual Light Affine Logic) by Baillot and Terui [3] and the STA (Soft Type Assignment) by Gaboardi and Ronchi Della Rocca [16]. Both systems are based on Light Logics, derived from the Linear Logic of Girard [17]. More precisely, types of DLAL are a proper subset of formulae of LAL (Light Affine Logic) by Asperti and Roversi [1], a simplified affine version of the Light Linear Logic of Girard [18], while types of STA are a proper subset of formulae of SLL (Soft Linear Logic) by Lafont [21]. The design of both systems is based on the transfer of the complexity properties from logics to terms, according to the proofs-as-programs approach inspired by the Curry–Howard isomorphism. Both characterize the polynomial functions, in

☆ Work partially supported by Istituto Nazionale di Alta Matematica "F. Severi" and by the MIUR COFIN project 2010FP79LR_007.
* Corresponding author.
E-mail addresses: debenede@di.unito.it (E. De Benedetti), ronchi@di.unito.it (S. Ronchi Della Rocca).

http://dx.doi.org/10.1016/j.ic.2015.12.012
0890-5401/

the sense that all and only the polynomial functions can be coded in such systems, according to the standard coding of functions by λ-terms. The logical inspiration of both systems is at the same time the key ingredient of their correctness and the responsible for their weak expressivity, since they can code few algorithms. From a typability point of view, in both systems the typable terms belong to a proper subset of the strongly normalizing terms.

A stronger expressive power could be achieved by enriching the language, and this approach has been followed by many authors. In particular, an extension of STA has been designed in [10], where some features like ML-polymorphism have been added to λ-terms, and a typed extension of DLAL has been proposed in [2], where a typed recursion has been introduced, besides other programming features.

Here we want to explore a different direction; namely, we want to preserve having pure λ-calculus as a programming language, but at the same time we want to design a system with stronger typability power, hoping to obtain, as side effect, also a gain in expressivity. The resulting STR (stratified) system is polynomial, in the previous sense, when a uniform coding scheme for data is considered; moreover, every strongly normalizing term is typable in STR, thus increasing in a significant way the typability power with respect to both DLAL and STA. In particular, STR is more expressive than STA, since a restricted form of iteration can be typed in the former, but not in the latter.

In STR types can be either linear or stratified. Linear types represent linear premises, in the sense of Linear Logic, and the operation of stratification is a sort of soft promotion. From a logical point of view, while in STA the promotion is a sort of multiple contraction for different copies of the same premise, here we can contract also premises having different types. This feature can no more be expressed in a logical way; indeed, STR is introduced as type assignment system without a pure logical counterpart.

In order to build STR we were inspired by intersection types [11]. Indeed, the relation between STA and STR reminds, in a very rough way, the relation between simple types and intersection types assignment system, the second being derived from the first, but allowing a variable to be assigned different types. The relation of the present work with intersection types is further discussed in the conclusion.

STR preserves the polynomial bound; indeed, the introduction of the intersection increases the typability power without increasing the computability power, as proved in [9]. In the relation between STA and STR, the same phenomenon happens: STR allows to type all the strongly normalizing terms, but it characterizes the same class of functions as STA, i.e. FPTIME. However the expressivity is increased, since bounded iteration cannot be expressed in STA, while in STR a restricted iteration construct can be typed.

The paper is organized as follows: in Section 2 we introduce the type assignment system STR and we prove that it enjoys the subject reduction property. In Section 3 we prove that STR characterizes strong normalization. In Section 4, we prove that STR is sound and complete with respect to FPTIME. In Section 5, we comment on the choice of stratified types with respect to intersection types. Finally, in Section 6, we conclude with some technical observations on the use of intersection types for quantitative purposes.

2. The STR Type Assignment system

In this section we introduce the type assignment system for λ-calculus named STR, based on the notion of *stratification* of types, and we prove that it enjoys subject reduction.

Definition 1.

i. The set Λ of *terms* is defined by the following syntax:

$$\mathtt{M} ::= \mathtt{x} \mid \lambda\mathtt{x}.\mathtt{M} \mid \mathtt{M}\mathtt{M}$$

where x ranges over a countable set of variables $\mathcal{X}$.
FV(M) denotes the set of free variables of the term M. Terms are considered modulo α-equivalence; moreover, bound variables are assumed to be all distinct and different from free ones.
The symbol $\equiv$ denotes the identity on terms, modulo renaming of bound variables.
A *(term) context* is generated by the same grammar, starting from a constant [.] (the hole), in addition to variables. Term contexts are denoted by C[.], and C[M] denotes the result of plugging M into every occurrence of [.] in C[.]. Observe that, as usual, the plugging operation allows the capture of free variables.

ii. The reduction relation $\longrightarrow_\beta$ is the contextual closure of the rule $(\lambda\mathtt{x}.\mathtt{M})\mathtt{N} \to \mathtt{M}[\mathtt{N}/\mathtt{x}]$, where the substitution $\mathtt{M}[\mathtt{N}_1/\mathtt{x}_1, \ldots, \mathtt{N}_n/\mathtt{x}_n]$, also denoted by $\mathtt{M}[\mathtt{N}_i/\mathtt{x}_i]_{i=1}^n$, is the capture-free substitution of $\mathtt{N}_i$ to all the free occurrences of $\mathtt{x}_i$ in M ($1 \le i \le n$). The relation $\overset{*}{\longrightarrow}_\beta$ is the reflexive and transitive closure of $\longrightarrow_\beta$.

iii. The set of pre-types is defined by the following syntax:

$$\mathtt{A} ::= \mathtt{a} \mid \sigma \multimap \mathtt{A} \mid \forall\mathtt{a}.\mathtt{A} \quad \text{(linear pre-types)}$$

$$\sigma ::= \mathtt{A} \mid \{\underbrace{\sigma, \ldots, \sigma}_{n}\} \quad \text{for } n > 0 \text{ (stratified pre-types)}$$