Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 136 (2005) 23–42
www.elsevier.com/locate/entcs

On Type Inference in the Intersection Type Discipline

Gérard Boudol and Pascal Zimmer[1,2]

INRIA Sophia Antipolis, BP93
06902 Sophia Antipolis Cedex, France

Abstract

We introduce a new unification procedure for the type inference problem in the intersection type discipline. We show that unification exactly corresponds to reduction in an extended λ-calculus, where one never erases arguments that would be discarded by ordinary β-reduction. We show that our notion of unification allows us to compute a principal typing for any strongly normalizing λ-expression.

Keywords: λ-calculus, type systems, type inference, unification

1 Introduction

Type inference – say, for any λ-calculus based model –, as it is now presented in textbooks (see for instance [18], p. 136), generally proceeds as follows:

1. Assign a type to the expression and each subexpression. For any compound expression or variable, use a type variable.

2. Generate a set of constraints on types, reflecting the fact that, if a function is applied to an argument, then the type of the argument must agree with the type of the domain of the function.

3. Solve these constraints.

[1] email: Gerard.Boudol@sophia.inria.fr
[2] email: Pascal.Zimmer@sophia.inria.fr

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2005.06.016

This design of a type inference algorithm was first (as far as we can see) proposed by J. Morris in his thesis [20]. At the first step of this procedure, a decision has to be taken, in order to build the type of a function, that is an abstraction $\lambda x M$: in which way do we consider the collection of type variables $t_1, \ldots, t_m$ assigned to the various occurrences of x in M as a type? There are various possibilities, which are not unrelated:

- Simple (monomorphic, possibly recursive) types: a variable x has only one type. That is, one has the constraint that the t_i's are equal (with or without "occur check").
- Generalized (polymorphic) types: the constraint here is that x is only used in M with types which are *instances* of the type of the domain of $\lambda x M$.
- Intersection types: the collection $t_1, \ldots, t_m$ is considered as a type, interpreted as the *conjunction* of the t_i's.
- Subtyping: x is only used in M with types which are *subtypes* of the type of the domain of $\lambda x M$.

(The question, and thus the possible answers, would be different regarding the "let" construct, that is $(\lambda x M N)$, where the abstraction $\lambda x M$ does not have to be explicitly typed, see for instance [11,17].)

In this paper we are interested in type inference for the intersection type discipline, introduced by Coppo and Dezani [8], and independently by Pottinger [21] (see [2,4] for a complete review of various systems with intersection types). There is no algorithm for deciding typability in this system, called "system $\mathcal{D}$" in [16], since this is equivalent to strong normalizability. However, one can compute a *principal* typing for any typable expression [9,16,24], that is a typing from which any other typing for the given expression can be derived, by means of suitable operations, among which the most important one is *expansion* (for an explanation of this notion, see for instance [3]). Type inference can be achieved by normalizing the expression, and then typing the normal form, but obviously this cannot be extended to a language where one may wish to type non-terminating programs. Ronchi proposed in [23] a direct procedure, based on a generalized unification mechanism. This was later revisited by Kfoury [13], and then Kfoury and Wells [14], who used explicit *expansion variables*, in order to provide a better understanding of the operation of expansion, and showed that type inference is decidable for subsystems with a bounded rank restriction.

In this paper we introduce a new way of solving the typing constraints that arise from type inference for intersection types. To give an idea of our generalized unification procedure, let us recall that the constraints to solve