On Explicit Substitutions and Names
(Extended Abstract)

Eike Ritter and Valeria de Paiva *

School of Computer Science, University of Birmingham

Abstract. Calculi with explicit substitutions have found widespread acceptance as a basis for abstract machines for functional languages. In this paper we investigate the relations between variants with de Bruijn-numbers, with variable names, with reduction based on raw expressions and calculi with equational judgements. We show the equivalence between these variants, which is crucial in establishing the correspondence between the semantics of the calculus and its implementations.

1 Introduction

Explicit substitution calculi (or λσ-calculi for short) first appeared in a seminal paper by Abadi et al. [1]. The basic idea is that instead of having substitutions as a meta-level operation, as in traditional λ-calculus, we should make them part of the object-level calculus. The advantages of this approach are twofold. Firstly, it makes it possible to design much more efficient abstract machines as we are allowed to delay substitutions, and secondly it makes it much easier to prove them correct since the calculus and its implementation are closer.

There are several variants of calculi with explicit substitutions. Some of these variants are geared towards semantics [15], [3], others are derived with implementations in mind [9], [8], [2]. Rather than listing all variants, we explain in this paper what we take to be the principal differences between them. This way we describe what appears at first sight as various “design choices” for lambda-calculi. But we then justify why we have to develop calculi for each possible choice if we want to prove semantics and syntax equivalent. Moreover, by using the context handling of type theory as a guide, we are able to define a confluent calculus with explicit substitutions and names—something that Abadi et al. were not able to do.

1.1 Equations first versus Reductions first

There are two main approaches when defining typed λ-calculi with or without explicit substitutions. The first one, in the spirit of Martin Lof’s type theory [10], defines the calculus with equations-in-context. Reduction is then a derived

*Research supported under the EPSRC project no. GR/L28296, x-SLAM: The Explicit Substitutions Linear Abstract Machine.
notion, obtained by orienting the equations. The second approach considers the set of typed terms as a subset of the set of raw terms, and hence reduction is defined on raw terms, which are not necessarily well-formed. Equality is now the derived notion, namely it is the symmetric and transitive closure of the relation generated by the reduction rules.

The first approach is required when giving semantics to $\lambda$-calculi because only well-formed objects have a meaning. The second approach avoids the need to check for well-formedness during reduction, which is incorporated in the first approach. As a consequence, this approach is well-suited for implementations, but a semantics for terms can only be given by showing the equivalence of this presentation to the Martin Löf-style presentation. Whereas this equivalence is easy to prove in the case of the simply-typed $\lambda$-calculus (and hence it is not really necessary to differentiate between the two approaches in this case), the difference becomes crucial as soon as we add, for example, dependent types [14]. This difference becomes crucial again when we consider calculi with explicit substitutions.

This paper presents calculi for both approaches and shows their equivalence (see section 3). This is because we want to connect the implementation, which is based on the second approach, with the semantics, which is based on the first approach.

### 1.2 Typed versus untyped calculi

There are typed and untyped calculi with explicit substitutions, both of which are presented already in [1]. The typing rules enforce two different restrictions: firstly, they eliminate expressions with misuse of variables, e.g., ones where we try to substitute two different terms for the same variable simultaneously. Secondly, they ensure that the only well-typed $\lambda$-terms are the ones of the simply-typed $\lambda$-calculus.

### 1.3 Names versus de Bruijn numbers

Another important kind of choice the designer of a explicit substitution $\lambda$-calculus can make concerns the difference between variable names and de Bruijn numbers. De Bruijn numbers were initially considered, as an implementational trick for Automath: instead of using variables like $x, y, z$ de Bruijn proposed to use natural numbers (that correspond to the binding level of the variable), in such a way that a class of $\alpha$-congruent terms correspond to a single syntactic object. Hence two expressions with variable names are $\alpha$-equivalent if and only if the corresponding terms with de Bruijn numbers are syntactically equal. More than simply an implementational trick, de Bruijn numbers are helpful when defining the semantics of the calculus in question. The point is that a de Bruijn-number $n$ corresponds exactly to the $n$-th projection $A_n \times \cdots \times A_1 \to A_n$.

---

1 The equivalence proofs can still be done [6], but some of the required properties of the type theories, like confluence and subject reduction, are very hard to establish.