A meta-language for typed object-oriented languages

Giuseppe Castagna*

LIENS(CNRS)-DMI
45 rue d'Ulm, 75005 Paris. FRANCE
e-mail: castagna@dmi.ens.fr

Abstract. In [3] we defined the $\lambda\&$-calculus, a simple extension of the typed $\lambda$-calculus to model typed object-oriented languages. To develop a formal study of type systems for object-oriented languages we define, in this paper, a meta-language based on $\lambda\&$ and we show by a practical example how to use it to prove properties of a language. To this purpose we define a toy object-oriented language and its type-checking algorithm; then we translate this toy language into our meta-language. The translation gives the semantics of the toy language and a theorem on the translation of well-typed programs proves the correction of the type-checker of the toy language.

As an aside we also illustrate the expressivity of the $\lambda\&$-based model by showing how to translate existing features like multiple inheritance and multiple dispatch, but also by integrating in the toy language new features directly suggested by the model, such as first-class messages, a generalization of the use of super and the use of explicit coercions.

1 Introduction

In [3] we introduced the $\lambda\&$-calculus. It is a simple extension of the typed lambda calculus to deal with overloaded functions, subtyping and dynamic binding. The main motivation of its definition was to give a kernel calculus possessing the key properties of object-oriented programming, in the line of some ideas of [6]. In the same paper we showed how this calculus could be intuitively used to model some features of object-oriented programming. It resulted that such a calculus yields a model orthogonal to the ones proposed in the literature so far. Thus we returned to object-oriented programming and we reviewed it in the light of the model arising from the $\lambda\&$-calculus. The experiment was surprising since we were able to deal with some features (such as multiple dispatch or the extension of the set of methods of a certain class) and introduce new ones (as first class messages or a generalization of "super") the usual models could not.

However, $\lambda\&$ is inadequate for a formal study of the properties of real object-oriented languages, and it was not meant for this: it is a calculus not a meta-language; thus, even if it possesses the key mechanisms to model object-oriented features, it cannot be used to "reason about" (i.e. to prove properties of) an object-oriented language.

For this reason in this paper we define a meta-language (i.e. a language to reason about —object-oriented— languages) that we call $\lambda_{\text{object}}$. This

* Supported by grant no. 203.01.56 of the Consiglio Nazionale delle Ricerche, Comitato Nazionale delle Scienze Matematiche, Italy, to work at LIENS

2 In this case the prefix "meta" is used w.r.t. the object-oriented languages
language is still based on the key mechanisms of $\lambda$k (essentially, overloading and dynamic binding) but it is enriched by those features (like commands to define new types, to work on their representations, to handle the subtyping hierarchy, to change the type of a term or to modify the discipline of dispatching etc.) that are necessary to reproduce the constructs of a programming language and that $\lambda$k lacks for.

We also show, by a practical example, how to use $\lambda$.object to prove properties of an object-oriented language. To this purpose we define a simple toy object-oriented language (a mix of Objective-C and CLOS constructs) and an algorithm to type-check its programs. We then translate the programs of the toy object-oriented language into $\lambda$.object. We prove that every well typed program of the former is translated into a well typed program of the latter; since the latter enjoys the subject-reduction property, it implies that the reduction of the translated program never goes wrong on a type error; in particular this proves the correction of the type-checker for the toy language.

The paper is organized as follows: section 2 gives an informal description of the toy language and of its type discipline. In section 3 we briefly summarize the $\lambda$&-calculus. In section 4 we describe $\lambda$.object: we give its operational semantics, a type-checker and prove the subject reduction theorem. In section 5 we hint the translation and we prove the correction of the type discipline for the toy language. For space reasons we cannot give a detailed description of all the systems. All the details and the precise connection between $\lambda$k and $\lambda$.object will be included in the author's PhD. thesis.

2 The toy language

2.1 Message passing

There exist many syntaxes for messages; in our toy language message-expressions are enclosed in square brackets: \([\text{receiver message}]\). There are two ways to model message passing. One is to consider an object as a record of methods and message passing as dot selection (e.g. in Eiffel; see [8]). The other is to consider message passing as functional application where the message is the function and the receiver is the argument (as in CLOS; see [7]). In this paper we choose this second solution. Though the fact that a method belongs to a specific object (more precisely to a specific class of objects) implies that message passing is a mechanism different from the usual functional call (i.e. $\beta$-reduction). In our approach the main characteristics that distinguish messages from functions are:

Overloading: Two objects may respond differently to the same message. For instance, the code executed when sending a message inverse to an object representing a matrix will be different from the one executed when the same message is sent to an object representing a real number. But the same message behaves uniformly on objects of the same kind (e.g. on all objects of class matrix). This feature is known as overloading since we overload the same operator (in this case inverse) by different operations; the actual operation depends on the type of the operands. Thus messages are identifiers of overloaded functions and in message passing the receiver is the first argument of an overloaded function, i.e. the one on whose type is based the selection of the code to be executed. Each method constitutes a branch (i.e. a code or operation) of the overloaded function referred by the message it is associated to.

Dynamic binding: The second crucial difference between function application and message passing is that a function is bound to its meaning at compile time