Abstract. Labeled types and a new relation between types are added to the lambda calculus of objects as described in [6]. This relation is a trade-off between the possibility of having a restricted form of width subtyping and the features of the delegation-based language itself. The original type inference system allows both specialization of the type of an inherited method to the type of the inheriting object and static detection of errors, such as 'message-not-understood'. The resulting calculus is an extension of the original one. Type soundness follows from the subject reduction property.

1 Introduction

Object-oriented languages can be classified as either class-based or delegation-based languages. In class-based languages, such as Smalltalk [4] and C++ [5], the implementation of an object is specified by its class. Objects are created by instantiating their classes. In delegation-based languages, objects are defined directly from other objects by adding new methods via method addition and replacing old methods bodies with new ones via method override. Adding or overriding a method produces a new object that inherits all the properties of the original one. In this paper we consider the delegation-based axiomatic model developed by Fisher, Honsell and Mitchell, and, in particular, we refer to the model in [6] and [7]. This calculus offers:

- a very simple and effective inheritance mechanism,
- a straightforward mytype method specialization,
- dynamic lookup of methods, and
- easy definition of binary methods.

The original calculus is essentially an untyped lambda calculus enriched with object primitives. There are three operations on objects: method addition (denoted by \( e_1 \leftarrow m=e_2 \)) to define methods, method override \( (e_1 \leftarrow m=e_2) \) to re-define methods, and method call \( e \leftarrow m \) to send a message \( m \) to an object \( e \). In the system of [6], the method addition makes sense only if method \( m \) does not occur in the object \( e \), while method override can be done only if \( m \) occurs in \( e \). If the expression \( e_1 \) denotes an object without method \( m \), then \( e_1 \leftarrow m=e_2 \) denotes a new object obtained from \( e_1 \) by adding the method body \( e_2 \) for \( m \).
When the message \( m \) is sent to \( \langle e_1 \leftarrow m=e_2 \rangle \), the result is obtained by applying \( e_2 \) to \( \langle e_1 \leftarrow m=e_2 \rangle \) (similarly for \( \langle e_1 \leftrightarrow m=e_2 \rangle \)).

This form of self-application allows to model the special symbol \textit{self} of object oriented languages directly by lambda abstraction. Intuitively, the method body \( e_2 \) must be a function and the first actual parameter of \( e_2 \) will always be the object itself. The type system of this calculus allows methods to be specialized appropriately as they are inherited.

We consider the type of an object as the collection of the types of its methods. The intuitive definition of the \textit{width} subtyping then is: \( \sigma \) is a subtype of \( \tau \) if \( \sigma \) has more methods than \( \tau \). The standard subsumption rule allows to use an object of type \( \sigma \) in any context expecting an object of type \( \tau \). In the original object calculus of [6], no \textit{width} subtyping is possible, because the addition of the method \( m \) to the object \( e \) is allowed if and only if \( m \) does not occur in \( e \). So, the object \( e \) could not be replaced by an object \( e' \) that already contains \( m \).

Moreover, it is not possible to have \textit{depth} subtyping, namely, to generalize the types of methods that appear in the type of the object, because with method override we can give type to an expression that produces run-time errors (a nice example of [1] is translated in the original object calculus in [8]).

In this paper, we introduce a restricted form of subtyping, informally written as \( \sigma \preceq \tau \). This relation is a \textit{width} subtyping, i.e., a type of an object is a subtype of another type if the former has more methods than the latter. Subtyping is constrained by one restriction: \( \sigma \) is a subtype of another type \( \tau \) if and only if we can assure that the methods of \( \sigma \), that are not methods of \( \tau \), are not referred to by the methods also in \( \tau \). The restriction is crucial to avoid that methods of \( \tau \) will refer to the forgotten methods of \( \sigma \), causing a run-time error. The subtyping relation allows to forget methods in the type without changing the shape of the object; it follows that we can type programs that accept as actual parameters objects with more methods than could be expected. The information on which methods are used is collected by introducing \textit{labeled types}. A first consequence of this relation is that it can be possible to have an object in which a method is, via a new operation, added more than once. For this reason, we introduce a different symbol to indicate the method addition operation on objects, namely \( \langle e_1 \leftarrow \circ m=e_2 \rangle \).

The operation \( \leftarrow \circ \) behaves exactly as the method addition of [6], but it can be used to add the same method more than once. For example, in the object \( \langle \langle e_1 \leftarrow \circ m=e_2 \rangle \leftarrow \circ m=e_3 \rangle \),

the first addition of the method \( m \) is forgotten by the type inference system via a subsumption rule. Our extension gives the following (positive) consequences:

- objects with extra methods can be used in any context where an object with fewer methods might be used,
- our subtyping relation does not cause the shortcomings described in [1],
- we do not loose any feature of the calculus of [6].

We also extend the set of objects and we present an alternative operational semantics. Our evaluation rules search method bodies more directly and deal with possible errors. This semantics was inspired by [2], where the calculus is