Simple overloading for type theories

Peter Aczel
Departments of Mathematics and Computer Science
Manchester University

June 23, 1994

In this note I describe a simple disciplined scheme for overloading of functions in the type theories like those implemented in Alf, Coq or Lego. This form of overloading is intended to be useful in the formalisation of mathematics in these computer systems. To be specific I will focus on the Lego system.

Direct Overloading

Before describing the disciplined approach that I will call *overloading by inheritance* I will start by describing a direct version of overloading that is my understanding of an idea explained to me by Chet Murthy. Recall that in Lego a context may have a function definition of the form

$$f[x : A] = e$$

where f must be an identifier that is new to the context of the definition and cannot be redefined in the context. The idea for *direct overloading* is to allow a context to have several *overload definitions* for the same identifier f:-

$$\begin{array}{lll} \textsf{overload} & f[x : A_1] & = e_1 \\ & \vdots & \\ \textsf{overload} & f[x : A_n] & = e_n \end{array}$$

In a context with such overload definitions an occurrence of f should always be in an application

$$f\ a$$

and the overloading of f in that application should be resolved by choosing the most recent overload definition of f

$$\textsf{overload}\quad f[x : A_i] = e_i$$

such that $a : A_i$.

Any development that has been checked in an extension of Lego with direct overloading can be transformed into a development that will be checked in the unextended Lego by replacing each overload definition

$$\textsf{overload}\quad f[x : A_i] = e_i$$

by an ordinary definition

$$f_i[x : A_i] = e_i,$$

where each f_i must be a new identifier, and also by replacing each application occurrence fa of f by $f_i a$, where f_i is the new identifier used to resolve the overloading of f for that particular application occurrence. In this way we see that the extension of Lego with direct overloading is conservative over the unextended Lego.

Overloading by Inheritance

We now consider the more sophisticated and disciplined form of overloading that I am calling *overloading by inheritance*. The idea is to extend Lego by allowing class and method definitions to appear in contexts. Each method will be associated with a particular class but can be inherited by subclasses. Each class C will have an associated type $\hat{}C$ of the *instances* of C, and each method m on C will be given in terms of a function on $\hat{}C$. There will be a subclass relation on the finitely many classes defined in a context and for each subclass C' of a class C there will be a *coercion function* $\hat{}C' \to \hat{}C$, so that any function defined on $\hat{}C$ will induce a function defined on $\hat{}C'$. So each method on C will induce a function on the type $\hat{}C'$ of instances of any subclass C' of C.

We will see that each method definition on a class C of the overloading by inheritance extension of Lego will be translated into the direct overloading extension of Lego as a number of overload definitions of the identifier m, one for each definition of a subclass of C.

The classes defined in a context will form a *class forest*; i.e. a finite set of finite trees whose nodes are labelled with the classes. Associated with each class definition will be an implicit type definition

$$\hat{}C = T$$

where T is a type expression in the context of the class definition.

There are two forms of class definition depending on whether a root of a new tree of the class forest is being created or a new leaf of an already existing tree is being created. A *root class definition* has the form

$$C = \mathsf{rootclass} \quad T$$

and creates the root of a new tree of the class forest that is labelled with C and has the implicit type definition

$$\hat{}C = T.$$

A *child class definition* has the form

$$C = \mathsf{childclass} \quad C_0[x : T]b$$

and creates a new leaf, labelled with C, of the tree immediately above the node labelled with the previously defined class C_0 and also has the implicit type definition

$$\hat{}C = T.$$

The function $[x : T]b$ is the coercion function $\hat{}C \to \hat{}C_0$.

If, in a tree of the class forest of a context, the class C' appears at or above the class C (where we take trees to grow upwards), then C' is a *subclass* of C and the coercion function $\hat{}C' \to \hat{}C$ is obtained by composing the coercion functions between each child class and its parent on the path from C' to C.

A *method definition* for a class C has the form

$$m = \textsf{method} \quad [x : \hat{}C]e$$

where, for $x : \hat{}C$, e is a term, and m is a new identifier. In a context where such a method definition occurs m should be allowed in an application

$$m\ a$$

provided that a can be determined to be an instance $a : \hat{}C'$ of some subclass C' of C, and should be treated as defined to be $e[a'/x]$, where a' is the result of applying the coercion $\hat{}C' \to \hat{}C$ to a.

I now describe how to translate a development in the extension of Lego with overloading by inheritance to the previous extension of Lego with direct overloading. Each class definition

$$C = ...T...$$

should be replaced by the type definition

$$C = T,$$

with all occurrences of $\hat{}C$ replaced by C. At the same time each method definition

$$m = \textsf{method} \quad [x : \hat{}C]e$$

should be replaced by an overload definition

$$\textsf{overload} \quad m[x : C] = e$$

and whenever a subclass C' of C was defined there should be an additional overload definition

$$\textsf{overload} \quad m[x : C'] = e[cx/x]$$

where c is the coercion function $\hat{}C' \to \hat{}C$.