A notion of class for Type Theory
DRAFT

Peter Aczel and Gilles Barthe
Manchester

August 16, 1993

Introduction

In order to conveniently formalise mathematics in a type theory it would be useful to have incorporated into the type theory a good theory of classes in something like the sense of class that there is in object oriented programming. To start with we will restrict attention to classes with non-multiple inheritance. But we would eventually also like to deal with multiple inheritance as well.

We want classes to form the nodes of a tree, with a class $Triv$ at the root and every other class having a unique parent that it has been constructed from. Given a class C, it and its descendents in the tree are the *subclasses* of C. We write $C' \leq C$ if C' is a subclass of C.

With each class C should be associated the type $C.ob$ of the *objects* of C. If $C' \leq C$ then there should be a function[1]

$$down_C^{C'} : C'.ob \to C.ob.$$

Given a class C and a C-family[2] F of types, if[3] $p : \{a : C.ob\}F\ a$ then we want to associate with p an *F-method for C*. We will write p^* for this method, leaving its dependance on C and F implicit. We want a type $(Method\ C\ F)$ for such methods.

The key idea is that if q is any F-method and $a : C'.ob$, where $C' \leq C$, then $(q\ a)$ should be a well-formed term of type $F\ (down_C^{C'}\ a)$. Moreover if q is p^* then we want $(q\ a)$ to convert to $p\ (down_C^{C'}\ a)$. What we have here is that the notion of a method involves an implicit polymorphism. When applying a method for a class C to an object of a subclass, the subclass does not need to be mentioned explicitly.

Classes in Algebra

Here is an example from algebra to illustrate the intended application of a theory of classes to the formalisation of mathematics.

[1]As trees grow upwards, *down* is directed downwards towards the root
[2]i.e. $F\ a$ is a type for each $a : C.ob$
[3]We use Lego syntax for type theory

It is natural to use something like $+_A$ for the group operation on any abelian group A. It is also natural to still use the same expression when A is a ring, field, vector space, etc... But when we routinely formalise these notions in a type theory we find that we get, say, an operation *plus* A for each abelian group A, but if R is a ring then the plus operation on R must be written *plus* $(RingToAbgroup\ R)$, where $RingToAbgroup$ is an operation that applies to any ring and obtains from it the underlying abelian group structure. This makes the syntax too heavy! It would be better to form the *class* of abelian groups and the subclasses of rings, fields, vector space, etc... and obtain *plus* as a method. In this way we can keep to the informal style.

Note that in informal algebra there is a further level of polymorphism that is very convenient, but should not be confused with that concerning the above use of classes. Suppose that we have a ring R and elements x, y of R. Then we would prefer to write just $x + y$ rather than $x +_R y$ when there is no confusion. There might be confusion because there might be another ring R' with the same type of elements, but where $+_R$ and $+_{R'}$ are different operations. This kind of implicit polymorphism can be dealt with in the Lego system. For example the polymorphic addition operation on the type of rings should be given the type

$$\{R|Ring\}R.el \rightarrow R.el \rightarrow R.el$$

where $Ring$ is the type of rings and if R is a ring then $R.el$ is the type of elements of R. Here the implicitness of the polymorphism is indicated by using $R|Ring$ rather than $R : Ring$. Now if $x, y : R.el$, where R is a ring and *plus* has the above type then we can write *plus* $x\ y$ and hope that the Lego system will be able to recover the ring R.

It has seemed to us that a suitable notion of class, as we have indicated above might be simulated in Lego by exploiting the implicit polymorphism lartethat is already available. But, so far, we are not sure that we have got a satisfactory simulation. We have in mind a notion of class where a method will always apply in a predictable way to any object of any subclass of the original class of the method. But unfortunately we have not seen how to predict the behaviour of the algorithm that Lego uses to type check expressions involving implicit polymorphism, so that our attempted simulations have inherited the same unpredictability.

Constructing Classes

So far our notion of class is incompletely specified, as we have not stipulated how classes are to be constructed, apart from the trivial class $Triv$ at the root of the tree of classes. We take $Triv.ob$ to be the unit type with a single element $triv : Triv.ob$. So this class, $Triv$ has just one object, containing no information.

Given a class C and a C-family of types P we want to have a new class

$$C' = (newclass\ C\ P)$$

that is an immediate subclass of C; i.e. a child of C in the tree of classes. We want the objects of C' to have the form

$$(a, b)',$$

where $a : C.ob$ and $b : (P\ a)$. Here $(a, b)'$ is some kind of ordered pair that we may want to include some additional information not explicitly indicated. So $C'.ob$ is something like the sigma type $< a : C.ob > (P\ a)$. The previously mentioned operation $down_C^{C'}$ is, in this case of an immediate subclass, given by

$$down_C^{C'}\ (a, b)' = a.$$

In general the down operations can be obtained by iterated projections on the first coordinate.