

First-Class Type Classes

Matthieu Sozeau¹ and Nicolas Oury²

¹ Univ. Paris Sud, CNRS, Laboratoire LRI, UMR 8623, Orsay, F-91405

INRIA Saclay, ProVal, Parc Orsay Université, F-91893

sozeau@lri.fr

² University of Nottingham

np0@cs.nott.ac.uk

Abstract. Type Classes have met a large success in HASKELL and ISABELLE, as a solution for sharing notations by overloading and for specifying with abstract structures by quantification on contexts. However, both systems are limited by second-class implementations of these constructs, and these limitations are only overcome by ad-hoc extensions to the respective systems. We propose an embedding of type classes into a dependent type theory that is first-class and supports some of the most popular extensions right away. The implementation is correspondingly cheap, general and integrates well inside the system, as we have experimented in COQ. We show how it can be used to help structured programming and proving by way of examples.

1 Introduction

Since its introduction in programming languages [1], *overloading* has met an important success and is one of the core features of object-oriented languages. Overloading allows to use a common name for different objects which are *instances* of the same type schema and to automatically select an *instance* given a particular type. In the functional programming community, overloading has mainly been introduced by way of *type classes*, making ad-hoc polymorphism less ad hoc [17].

A *type class* is a set of functions specified for a parametric type but defined only for some types. For example, in the HASKELL language, it is possible to define the class **Eq** of types that have an equality operator as:

```
class Eq a where (==) :: a → a → Bool
```

It is then possible to define the behavior of == for types whose equality is decidable, using an instance declaration:

```
instance Eq Bool where x == y = if x then y else not y
```

This feature has been widely used as it fulfills two important goals. First, it allows the programmer to have a uniform way of naming the same function over different types of data. In our example, the *method* == can be used to compare any type of data, as long as this type has been declared an instance of the **Eq**

class. It also adds to the usual Damas–Milner [4] parametric polymorphism a form of ad-hoc polymorphism. E.g, one can constrain a polymorphic parameter to have an instance of a type class.

```
=/ :: Eq  $a \Rightarrow a \rightarrow a \rightarrow$  Bool
 $x =/ y = \text{not } (x == y)$ 
```

Morally, we can use `==` on x and y because we have supposed that a has an implementation of `==` using the type class *constraint* '`Eg a =>`'. This construct is the equivalent of a functor argument specification in module systems, where we can require arbitrary structure on abstract types.

Strangely enough, overloading has not met such a big success in the domain of computer assisted theorem proving as it has in the programming language community. Yet, overloading is very present in non formal—pen and paper—mathematical developments, as it helps to solve two problems:

- It allows the use of the same operator or function name on different types. For example, `+` can be used for addition of both natural and rational numbers. This is close to the usage made of type classes in functional programming. This can be extended to proofs as well, overloading the *reflexive* name for every reflexivity proof for example (see §3.4).
- It shortens some notations when it is easy to recover the whole meaning from the context. For example, after proving (M, ε, \cdot) is a *monoid*, a proof will often mention the monoid M , leaving the reader implicitly inferring the neutral element and the law of the monoid. If the context does not make clear which structure is used, it is always possible to be more precise (an extension known as named instances [9] in HASKELL). Here the name M is overloaded to represent both the carrier and the (possibly numerous) structures built on it.

These conventions are widely used, because they allow to write proofs, specifications and programs that are easier to read and to understand.

The COQ proof assistant can be considered as both a programming language and a system for computer assisted theorem proving. Hence, it can doubly benefit from a powerful form of overloading.

In this paper, we introduce *type classes* for COQ and we show how it fulfills all the goals we have sketched. We also explain a very simple and cheap implementation, that combines existing building blocks: *dependent records*, *implicit arguments*, and *proof automation*. This simple setup, combined with the expressive power of dependent types subsumes some of the most popular extensions to HASKELL type classes [2,8].

More precisely, in this paper,

- we first recall some basic notions of type theory and some more specific notions we are relying on (§2) ;
- then, we define *type classes* and *instances* before explaining the basic design and implementation decisions we made (§3) ;
- we show how we handle *superclasses* and *substructures* in section 4 ;