First-Class Type Classes

Matthieu Sozeau[1] and Nicolas Oury[2]

[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
npo@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 overcomed 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` (`==`) :: $a \rightarrow a \rightarrow$ **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**

O. Ait Mohamed, C. Muñoz, and S. Tahar (Eds.): TPHOLs 2008, LNCS 5170, pp. 278–293, 2008.

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.

$$\texttt{=/=} :: \textbf{Eq}\ a \Rightarrow a \to a \to \textbf{Bool}$$
$$x\ \texttt{=/=}\ y = \mathsf{not}\ (x\ \texttt{==}\ y)$$

Morally, we can use == on x and y because we have supposed that a has an implementation of == using the type class *constraint* '**Eq** $a \Rightarrow$'. 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 $(M, \varepsilon, \cdot)$ 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 ;

- we present two detailed examples (§5): a library on monads and a development of category theory ;
- we discuss the implementation and possible extensions to the system (§6) ;
- finally, we present related and future work (§7).

Though this paper focuses mainly on the Coq proof assistant, the ideas developed here could easily be used in a similar way in any language based on Type Theory and providing both dependent records and implicit arguments.

2 Preliminaries

A complete introduction to Type Theory and the Coq proof assistant is out of the scope of this article. The interested reader may refer to the reference manual [16] for an introduction to both. Anyway, we recall here a few notations and concepts used in the following sections.

2.1 Dependent Types

Coq's type system is a member of the family of Type Theories. In a few words, it is a lambda-calculus equipped with *dependent types*. It means that types can depend upon values of the language. This allows to refine the type of a piece of data depending on its content and the type of a function depending on its behavior. For example, it is possible, using Coq, to define the type of the lists of a given size n and to define a function *head* that can only be applied to lists of a strictly positive size.

Indeed, the type can specify the behavior of a function so precisely, that a type can be used to express a *mathematical property*. The corresponding program is then the *proof* of this property. Hence, the Coq system serves both as a programming language and a proof assistant, being based on a single unifying formalism.

2.2 Dependent Records

Among the different types that can be defined in Coq, we need to explain more precisely *dependent records*, as they serve as the basis of our implementation of type classes. A dependent record is similar to a record in a programming language, aggregating different fields, each possessing a *label*, a *type* and a *value*. Dependent records are a slight generalization of this as the type of a field may refer to the value of a preceding field in the record.

For example, for a type A and a property P on A, one can express the property *"there exist an element a in a type A such that* $(P\ a)$ *holds"* using a dependent record whose:

- First field is named *witness* and has type A.
- Second field is named *proof* and is of type P *witness*. The type of this second field *depends* on the value of the first field *witness*.

Another common use of dependent records, that we are going to revisit in the following, is to encode *mathematical structures* gluing some *types*, *operators* and *properties*. For example, the type of monoids can be defined using a dependent record. In Coq syntax:

```
Record Monoid := {
  carrier : Type ;
  neutral : carrier ; op : carrier → carrier → carrier ;
  neutral_left : ∀ x, op neutral x = x ; ··· }
```

We can observe that the neutral element and the operation's types refer to carrier, the first field of our record, which is a type. Along with the type and operations we can also put the monoid laws. When we build a record object of type **Monoid** we will have to provide proof objects correspondingly. Another possibility for declaring the **Monoid** data type would be to have the carrier type as a parameter instead of a field, making the definition more polymorphic:

```
Record Monoid (carrier : Type) := {
  neutral : carrier ; op : carrier → carrier → carrier ; ··· }
```

One can always abstract fields into parameters this way. We will see in section §4 that the two presentations, while looking roughly equivalent, are quite different when we want to specify sharing between structures.

2.3 Implicit Arguments

Sometimes, because of type dependency, it is easy to infer the value of a function argument by the context where it is used. For example, to apply the function id of type $\forall\ A : \mathtt{Type},\ A \to A$ one has to first give a type X and then an element x of this type because of the explicit polymorphism inherent to the Coq system. However, we can easily deduce X from the type of x and hence find the *value* of the first argument automatically if the second argument is given. This is exactly what the implicit argument system does. It permits to write shortened applications and use unification to find the values of the implicit arguments. In the case of id, we can declare the first argument as implicit and then apply id as if it was a unary function, making the following type-check:

```
Definition foo : bool := id true.
```

One can always explicitly specify an implicit argument at application time, to disambiguate or simply help the type-checker using the syntax:

```
Definition foo : bool := id (A:=bool) true.
```

This is very useful when the user himself specifies which are the implicit arguments, knowing that they will be resolved in some contexts (due to typing constraints giving enough information for example) but not in others. Both mechanisms of resolution and disambiguation will be used by the implementation of type classes. Implicit arguments permit to recover reasonable (close to ML) syntax in a polymorphic setting even if the system with non-annotated terms does not have a complete inference algorithm.

3 First Steps

We now present the embedding of type classes in our type theory beginning with a raw translation and then incrementally refine it. We use the syntax $\overrightarrow{x_n : t_n}$ to denote the ordered typing context $x_1 : t_1, \cdots, x_n : t_n$. Following the HASKELL convention, we use capitalized names for type classes.

3.1 Declaring Classes and Instances

We extend the syntax of COQ commands to declare classes and instances. We can declare a class **Id** on the types τ_1 to τ_n having members $\mathsf{f}_1 \cdots \mathsf{f}_m$ and an instance of **Id** using the following syntax:

$$\begin{array}{ll}
\texttt{Class}\ \mathbf{Id}\ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) := & \texttt{Instance}\ \mathbf{Id}\ t_1 \cdots t_n := \\
\quad \mathsf{f}_1 : \phi_1\ ; & \quad \mathsf{f}_1 := b_1\ ; \\
\quad \vdots & \quad \vdots \\
\quad \mathsf{f}_m : \phi_m. & \quad \mathsf{f}_m := b_m.
\end{array}$$

Translation. We immediately benefit from using a powerful dependently-typed language like COQ. To handle these new declarations, we do not need to extend the underlying formalism as in HASKELL or ISABELLE but can directly use existing constructs of the language.

A newly declared class is automatically translated into a record type having the index types as *parameters* and the same members. When the user declares a new instance of such a type class, the system creates a record object of the type of the corresponding class.

In the following explanations, we denote by $\Gamma_{\mathbf{Id}} \triangleq \overrightarrow{\alpha_n : \tau_n}$ the context of parameters of the class **Id** and by $\Delta_{\mathbf{Id}} \triangleq \overrightarrow{\mathsf{f}_m : \phi_m}$ its context of definitions. Quite intuitively, when declaring an instance $\overrightarrow{t_n}$ should be an instance of $\Gamma_{\mathbf{Id}}$ and $\overrightarrow{b_m}$ an instance of $\Delta_{\mathbf{Id}}[\overrightarrow{t_n}]$.

Up to now, we just did wrapping around the `Record` commands, but the declarations are not treated exactly the same. In particular, the projections from a type class are declared with particular implicit parameters.

3.2 Method Calls

When the programmer calls a type class method f_i, she does not have to specify which instance she is referring to: that's the whole point of overloading. However, the particular instance that is being used is bound to appear inside the final term. Indeed, class methods are encoded, in the final proof term, as applications of some projection of a record object representing the instance $\mathsf{f}_i : \forall \Gamma_{\mathbf{Id}}, \mathbf{Id}\ \overrightarrow{\alpha_i} \to \phi_i$. To be able to hide this from the user we declare the instance of the class **Id** as an implicit argument of the projection, as well as all the arguments corresponding to $\Gamma_{\mathbf{Id}}$.

For example, the method call x `==` y where x, y : **bool** expands to the application (`eq`) (?B:`Type`) (?e:**Eq** ?B) x y, where ?$A : \tau$ denotes unification variables

corresponding to implicit arguments, whose value we hope to discover during type checking. In this case $?B$ will be instantiated by **bool** leaving an **Eq bool** constraint.

Indeed, when we write a class method we expect to find the instance automatically by resolution of implicit arguments. However, it will not be possible to infer the instance argument using only unification as is done for regular arguments: we have to do a kind of proof search (e.g., here we are left with a **Eq** $?B$ constraint). So, after type-checking a term, we do constraint solving on the unification variables corresponding to implicit arguments and are left with the ones corresponding to type class constraints.

3.3 Instance Search

These remaining constraints are solved using a special purpose tactic that tries to build an instance from a set of declared instances and a type class constraint. Indeed, each time an instance is declared, we add a lemma to a database corresponding to its type. For example, when an instance of the class **Eq** for booleans is declared, a lemma of type **Eq bool** is created. In fact, any definition whose type has a conclusion of the form **Eq** τ can be used as an instance of **Eq**.

When searching for an instance of a type class, a simple proof search algorithm is performed, using the lemmas of the database as the only possible hints to solve the goal. Following our example, the remaining constraint **Eq bool** will be resolved if we have (at least) one corresponding lemma in our database. We will discuss the resolution tactic in section 6. Note that we can see the result of the proof search interactively in Coq by looking at the code, and specify explicitely which instance we want if needed using the $(A{:=}t)$ syntax.

3.4 Quantification

Finally, we need a way to parameterize a definition by an arbitrary instance of a type class. This is done using a regular dependent product which quantifies on implementations of the record type.

For convenience, we introduce a new binding notation [**Id** $\overrightarrow{t_n}$] to add a type class constraint in the environment. This boils down to specifying a binding $(\mathit{instance} : \textbf{Id}\ \overrightarrow{t_n})$ in this case, but we will refine this construct soon to attain the ease of use of the Haskell class binder. Now, all the ingredients are there to write our first example:

```
Definition neq (A : Type) [ Eq A ] (x y : A) : bool := not (x == y).
```

The process goes like this: when we interpret the term, we transform the [**Eq** A] binding into an $(\mathit{eqa} : \textbf{Eq}\ A)$ binding, and we add unification variables for the first two arguments of $(==) : \forall A : \mathtt{Type}, \textbf{Eq}\ A \to A \to A \to \textbf{bool}$. After type-checking, we are left with a constraint **Eq** A in the environment [A : Type, $\mathit{eqa} : \textbf{Eq}\ A$, $x\ y : A$], which is easily resolved using the *eqa* assumption. At this point, we can substitute the result of unification into the term and give it to the kernel for validation.

Implicit generalization. When writing quantifications on type classes, we will often find ourselves having to bind some parameters just before the type class

constraint, as for the A argument above. This suggests that it would be convenient to see type class binders as constructs binding their arguments. Indeed, when we give variables as part of the arguments of a class binder for **C**, we are effectively constraining the type of the variable as it is known from the $\Gamma_{\mathbf{C}}$ context. Hence we can consider omitting the external quantification on this variable and implicitly quantify on it as part of the class binder. No names need be generated and type inference is not more complicated. For example we can rewrite neq:

```
Definition neq [ Eq A ] (x y : A) : bool := negb (x == y).
```

This corresponds to the way type classes are introduced in HASKELL in prenex positions, effectively constraining the implicitly quantified type variables. If one wants to specify the type of A it is also possible using the notation $(A : \mathtt{Type})$ in the class binder. The arguments of class binders are regular contexts.

Hence the syntax also supports instantiated constraints like [**Eq** (**list** A)]. More formally, to interpret a list of binders we proceed recursively on the list to build products or lambda abstractions. For a type class binder $[\ \mathbf{C}\ \overrightarrow{t_n : \tau_n}\ ]$ we collect the set of free variables in the global environment extended by the terms $\overrightarrow{t_n}$, we extend the environment with them and the class binder and recursively interpret the rest of the binders in this environment.

General contexts. As hinted in the previous section, all the contexts we are manipulating are arbitrary. In particular this means that the parameters of a type class are not restricted to be types of kind Type (a strong limitation of ISABELLE's implementation). Any type constructor is allowed as a parameter, hence the standard **Monad** $(m : \mathtt{Type} \rightarrow \mathtt{Type})$ class causes no problem in our setting (see §5.1). In fact, as we are in a dependently-typed setting, one can even imagine having classes parameterized by terms like:

```
Class Reflexive (A : Type) (R : relation A) := reflexive : ∀ x : A, R x x.
```

We can instantiate this class for any A using Leibniz equality:

```
Instance Reflexive A (eq A) := reflexive x := refl_equal x.
```

Note that we are using implicit generalization again to quantify on A here. Now, if we happen to have a goal of the form R t t where t : T we can just apply the reflexive method and the type class mechanism will automatically do a proof search for a **Reflexive** T R instance in the environment.

4 Superclasses as Parameters, Substructures as Instances

We have presented a way to introduce and to use type classes and instances. The design relies on the strength of the underlying logical system. This strength makes it straightforward to extend this simple implementation with two key structuring concepts: inheritance and composition of structures.

4.1 Superclasses as Parameters

Inheritance permits to specify hierarchies of structures. The idea is to allow a class declaration to include type class constraints, known as superclasses in the Haskell terminology.

This kind of superclasses happens a lot in the formalization of mathematics, when the definition of a structure on an object makes sense only if the object already supports another structure. For example, talking about the class of functors from C to D only makes sense if C and D are categories or defining a group G may suppose that G is already a monoid. This can be expressed by superclass constraints:

```
Class [ C : Category, D : Category ] ⇒ Functor := ...
Class [ G : Monoid ] ⇒ Group := ...
```

As often when dealing with parameterization, there are two ways to translate inheritance in our system: either by making the superstructures *parameters* of the type class or regular *members*. However, in the latter case it becomes harder to specify relations between structures because we have to introduce equalities between record fields. For example, we may wish to talk about two **Functor**s F and G between two categories C and D to define the concept of an adjunction. If we have the categories on which a **Functor** is defined as members src and dst of the **Functor** class, then we must ensure that F and G's fields are coherent using equality hypotheses:

```
Definition adjunction (F : Functor) (G : Functor),
    src F = dst G → dst F = src G ...
```

This gets very awkward because the equalities will be needed to go from one type to another in the definitions, obfuscating the term with coercions, and the user will also have to pass these equalities explicitly.

The other possibility, known as Pebble-style structuring [13], encourages the use of parameters instead. In this case, superstructures are explicitly specified as part of the structure type. Consider the following:

```
Class [ C : Category obj hom, D : Category obj' hom' ] ⇒ Functor := ...
```

This declaration is translated into the record type:

```
Functor (obj : Type) (hom : obj → obj → Type) (C : Category obj hom)
  (obj' : Type) (hom' : obj' → obj' → Type) (D : Category obj' hom') := ...
```

We depart from Haskell's syntax where the parameters of a superclass have to be repeated as part of the type class parameters: they are automatically considered as parameters of the subclass in our case, in the order they appear in the superclass binders.

Note that we directly gave names to the type class binders instead of letting the system guess them. We will use these names C and D to specify sharing of structures using the regular dependent product, e.g.:

```
Definition adjunction [ C : Category obj hom, D : Category obj' hom',
  F : Functor obj hom C obj' hom' D,
  G : Functor obj' hom' D obj hom C ] := ...
```

The verbosity problem of using Pebble-style definitions is quite obvious in this example. However we can easily solve it using the existing implicit arguments technology and bind F and G using (F : **Functor** C D) (G : **Functor** D C) outside the brackets. One can also use the binding modifier '!' inside brackets to turn back to the implicit arguments parsing of Coq, plus implicit generalization.

This is great if we want to talk about superclasses explicitly, but what about the use of type classes for programming, where we do not necessarily want to specify hierarchies? In this case, we want to specify the parameters of the class only, and not the implementations of superclasses. We just need to extend the type-checking process of our regular binder to implicitly generalize superclasses. For example, when one writes [**Functor** *obj hom obj' hom'*], we implicitly generalize not only by the free variables appearing in the arguments but also by implementations C and D of the two category superclasses (the example is developed in section §5.2). This way we recover the ability to quantify on a subclass without talking about its superclasses but we still have them handy if needed. They can be bound using the (A:=t) syntax as usual.

4.2 Substructures as Instances

Superclasses are used to formalize *is-a* relations between structures: it allows, for example, to express easily that a group *is a* monoid with some additional operators and laws.

However, it is often convenient to describe a structure by its components. For example, one can define a ring by a carrier set $\mathcal{S}$ that *has* two components: a group structure and a monoid structure on $\mathcal{S}$. In this situation, it is convenient that an instance declaration of a ring automatically declares instances for its components: an instance for the group and an instance for the monoid.

We introduce the syntax :> for methods which are themselves type classes. This adds instance declarations for the corresponding record projections, allowing to automatically use overloaded methods on the substructures when working with the composite structure.

Remark. This is very similar to the coercion mechanism available in Coq [14], with which one can say that a structure is a subtype of another substructure. The source-level type-checking algorithm is extended to use this subtyping relation in addition to the usual conversion rule. It puts the corresponding projections into the term to witness the subtyping, much in the same way we fill unification variables with instances to build a completed term.

5 Examples

We now present two examples of type classes for programming and proving: an excerpt from a monad library and a development of some basic concepts of

category theory. Both complete developments [1] can be processed by the latest version of Coq which implements all the features we have described.

5.1 Monads

We define the **Monad** type class over a type constructor η having operations unit and bind respecting the usual monad laws (we only write one for conciseness). We use the syntax $\{a\}$ for specifying implicit arguments in our definitions.

```
Class Monad (η : Type → Type) :=
  unit : ∀ {α}, α → η α ;
  bind : ∀ {α β}, η α → (α → η β) → η β ;
  bind_unit_right : ∀ α (x : η α), bind x unit = x.
```

We recover standard notations for monads using Coq's notation system, e.g.:

```
Infix ">>=" := bind (at level 55).
```

We are now ready to begin a section to develop common combinators on a given **Monad** on η. Every definition in the section will become an overloaded method on this monad *mon*.

```
Section Monad_Defs.
  Context [ mon : Monad η ].
```

The following functions are directly translated from Haskell's prelude. Definitions are completely straightforward and concise.

```
  Definition join {α} (x : η (η α)) : η α := x >>= id.

  Definition liftM {α β} (f : α → β) (x : η α) : η β := a ← x ;; return (f a).
```

We can use the overloading support to write more concise proof scripts too.

```
  Lemma do_return_eta : ∀ α (u : η α), x ← u ;; return x = u.
  Proof. intros α u.
    rewrite ← (bind_unit_right u) at 2.
    rewrite (eta_expansion (unit (α:=α))).
    reflexivity.
  Qed.
```

Type classes are transparently supported in all constructions, notably fixpoints.

```
  Fixpoint sequence {α} (l : list (η α)) : η (list α) :=
    match l with
      | nil ⇒ return nil
      | hd :: tl ⇒ x ← hd ;; r ← sequence tl ;; return (x :: r)
    end.
```

They work just as well when using higher-order combinators.

```
  Definition mapM {α β} (f : α → η β) : list α → η (list β) :=
    sequence · map f.
```

[1] http://www.lri.fr/~sozeau/research/coq/classes.en.html

5.2 Category Theory

Our second example has a bit more mathematical flavor: we make a few constructions of category theory, culminating in a definition of the map functor on lists.

A **Category** is a set of objects and morphisms with a distinguished id morphism and a compose law satisfiying the monoid laws. We use a setoid (a set with an equivalence relation) for the morphisms because we may want to redefine equality on them, e.g. to use extensional equality for functions. The $\equiv$ notation refers to the overloaded equivalence method of **Setoid**.

```
Class Category (obj : Type) (hom : obj → obj → Type) :=
  morphisms :> ∀ a b, Setoid (hom a b) ;
  id : ∀ a, hom a a;
  compose : ∀ a b c, hom a b → hom b c → hom a c;
  id_unit_left : ∀ a b (f : hom a b), compose f (id b) ≡ f;
  id_unit_right : ∀ a b (f : hom a b), compose (id a) f ≡ f;
  assoc : ∀ a b c d (f : hom a b) (g : hom b c) (h : hom c d),
    compose f (compose g h) ≡ compose (compose f g) h.
```

We define a notation for the overloaded composition operator.

```
Notation " x · y " := (compose y x) (at level 40, left associativity).
```

A **Terminal** object is an object to which every object has a unique morphism (modulo the equivalence on morphisms).

```
Class [ C : Category obj hom ] ⇒ Terminal (one : obj) :=
  bang : ∀ x, hom x one ;
  unique : ∀ x (f g : hom x one), f ≡ g.
```

Two objects are isomorphic if the morphisms between them are unique and inverses.

```
Definition isomorphic [ Category obj hom ] a b : _ :=
  { f : hom a b & { g : hom b a | f · g ≡ id b ∧ g · f ≡ id a } }.
```

Using these definition, we can do a proof on abstract instances of the type classes.

```
Lemma terminal_isomorphic
  [ C : Category obj hom, ! Terminal C x, ! Terminal C y ] : isomorphic x y.
Proof.
  intros ; red.
  exists (bang x). exists (bang (one:=x) y).
  split.
    apply unique.
    apply (unique (one:=x)).
Qed.
```

We can define a **Functor** between two categories with its two components, on objects and morphisms. We keep them as parameters because they are an essential part of the definition and we want to be able to specify them later.

```
Class [ C : Category obj hom, D : Category obj' hom'] ⇒
  Functor (Fobj : obj → obj')
          (Fmor : ∀ a b, hom a b → hom' (Fobj a) (Fobj b)) :=
  preserve_ident : ∀ a, Fmor a a (id a) ≡ id (Fobj a);
  preserve_assoc : ∀ a b c (f : hom a b) (g : hom b c),
    Fmor a c (compose f g) ≡ Fmor b c g · Fmor a b f.
```

Let's build the **Functor** instance for the map combinator on lists. We will be working in the category of Coq types and functions. The arrow homset takes Types A and B to the $A \to B$ function space.

```
Definition arrow A B := A → B.
```

Here we build a setoid instance for functions which relates the ones which are pointwise equal, i.e: functional extensionality. We do not show the straightforward proofs accompanying each `Instance` declaration in the following. For example, we have to show that pointwise equality is an equivalence here.

```
Instance arrow_setoid : Setoid (arrow a b) :=
  equiv f g := ∀ x, f x = g x.
```

We define the category TYPE of Coq types and functions.

```
Instance TYPE : Category Type arrow :=
  morphisms a b := arrow_setoid ;
  id a x := x ;
  compose a b c f g := g · f.
```

It is then possible to create a **Functor** instance for map on **list**s.

```
Instance list_map_functor : Functor TYPE TYPE list map.
```

6 Discussion

The previous sections contain a description of the overall design and implementation of type classes and a few examples demonstrating their effective use in the current version of Coq. Our overall technical contribution is a realistic implementation of type classes in a dependently-typed setting. We have also introduced convenient syntactic constructions to build, use and reason about type classes. We now summarize our implementation choices and explore some further alleys in the design space of type classes for dependently–typed languages.

6.1 Overview

All the mechanisms we have described are implemented in the current version of Coq, as a rather thin layer on top of the implicit arguments system, the record implementation, and the type-checking algorithm. Each of these components had to be slightly enhanced to handle the features needed by type classes. We added the possibility of specifying (maximally inserted) implicit arguments

in toplevel definitions. We adapted the record declaration front-end to handle implicit arguments in `Class` and `Instance` definitions properly and permit giving only a subset of an instance's fields explicitly, leaving the rest to be proved interactively. We also have a binding of type classes with Russell [15] that uses its enriched type system and allows to handle missing fields as obligations.

6.2 Naming

We allow for a lot of names to be omitted by the user that we really need to have programmatically. To create fresh names is difficult and error-prone, and we tried to use as much information that the user gives as possible during name generation. We need more experiments with the system to know where we should require names to be given to get reasonably robust code and specifications.

One thing that we learned doing bigger examples is that naming instances is very useful, even if we don't know beforehand if we will need to refer to them explicitly to disambiguate terms. Disambiguation is rarely needed in practice but it is very helpful. We always allow it, so that one can specify a particular superclass instance when specifying a class binder or a specific type class parameter for any method if needed.

6.3 Computing

A common concern when using heavily parameterized definitions is how efficiently the code can be run. Indeed, when we define type classes having multiple superclasses or parameters and call a method it gets applied to a large number of arguments. The good thing is that both the interpreted evaluation functions and the virtual machine [5] of Coq are optimized to handle large applications. Also, we avoid trouble with proofs preventing reductions in terms that happen when using other styles of formalizations.

6.4 Searching

The current tactic that searches for instances is a port of the `eauto` tactic that does bounded breadth- or depth-first search using a set of applicable lemmas. The tactic is applied to all the constraints at once to find a solution satisfying all of them, which requires a backtracking algorithm. Our first experiments suggest that this is sufficient for handling type classes in the simply-typed version, à la Isabelle. However, with multiple parameters and arbitrary instances the problem becomes undecidable and we can only do our best efforts to solve as much problems as possible. We envisage a system to give more control over this part of the system, allowing to experiment with new ways of using type classes in a controlled environment.

An especially important related issue is the current non-deterministic choice of instances. The problem is not as acute as in Haskell because we have an *interactive* system and disambiguation is always possible. Also, in some cases (e.g., when using reflexive to search for a reflexivity proof) we simply do not care what instance is found, applying the principle of proof-irrelevance. However, in

a programming environment, one may want to know if ambiguity arises due to multiple instances for the same constraint appearing in the environment. This is undecidable with arbitrary instances, so one would also want to restrict the shape of allowed instances of some classes to ensure decidability of this test. We have not yet explored how to do this in Coq, but we think this could be internalized in the system using a reflection technique. We hope we could transport existing results on type inference for Haskell type classes in this setting.

Finally, we did not study the relation between type-inference and instance search formally as this has been done in the other systems. In our current setting, the two systems could be given separate treatment, as we do instance search only after the whole term is type-checked and unification of type variables is finished.

7 Related Work

The first family of related work concerns type classes in functional programming. Type classes were introduced and extended as part of the Haskell language. Some extensions has been proposed to make the type classes system more powerful, while still retaining good properties of type inference. Most of them solve problems that do not arise when introducing type classes in a dependently-typed language like Coq.

- The use of dependent records gives a straightforward support of multi-parameter, dependent type classes.
- The possibility to mix more liberally types and values could give us *associated types* with classes [2] (and with them functional dependencies [8]) for free, although we have not worked out the necessary changes on the system yet.
- Named instances [9] are a direct consequence of having a first-class encoding.

Some work has recently been done to extend Haskell with some partial support for types depending on types or values like Generalized Algebraic Data Types. These extensions often make use of type classes to write programs at the level of types. Our approach, using the full power of dependent types, allows to program types and values in the same functional language and is therefore much less contrived.

Another embedding of type classes using an implicit argument mechanism was done in the Scala object-oriented programming language [12] recently.

Isabelle. Haskell-style type classes are also an integral part of the Isabelle proof assistant [19]. Wenzel [18] has studied type classes in the Isabelle proof assistant, and recently F. Haftmann and him have given [6] a constructive explanation of the original axiomatic type classes in terms of locales [10]. This explanation is not needed in our case because the evidence passing is done directly in the core language. We always have the kernel's type-checker to tell us if the elaboration mechanism did something wrong at any point. However, the inference is clearly formalized in Isabelle whereas we have no similar result.

The relative (compared to Coq) lack of power of the higher-order logic in Isabelle/HOL makes it impossible to quantify on type constructors directly,

using ordinary constructs of the language. This prevents using the system to implement our library on monads for example. To overcome this problem, one has to work in an extension of HOL with Scott's Logic of Computable Functions. It is then possible to construct axiomatic type classes for type constructors using a domain-theoretic construction [7].

Dependent Records. Ample litterature exists on how to extend dependent type theories with records, e.g [3] gives a good starting point. We stress that our implementation did absolutely not change Coq's kernel and the associated type theory, and we would only benefit from more fully-featured records. We leave a thorough comparison with the Canonical Structure mechanism of Coq for future work.

7.1 Future Work

Classes and Modules. We could customize the existing extraction mechanism [11] from Coq to Haskell to handle type classes specially. However, this translation is partial as our type system is more powerful. It would be interesting to study this correspondence and also the connection between type classes and modules in the Coq system itself.

Examples. Some examples have already been developed and we have seen some improvements in the elegance and clarity of the formalization of mathematics and the ease of programming. Some further work needs to be done there.

On top of classes. The type class system gives a basis for communication between users and the system, by adhering to a common structuring principle. We are exploring ways to use this mechanism, for example by developing a new setoid rewriting tactic based on an internalization of signatures in the class system.

8 Conclusion

We have presented a type class system for the Coq proof assistant. This system is useful both for developing elegant programs and concise mathematical formalizations on abstract structures. Yet, the implementation is relatively simple and benefits from dependent types and pre–existing technologies. It could easily be ported on other systems based on dependent types.

References

1. Birtwistle, G.M., Dahl, O.-J., Myhrhaug, B., Nygaard, K.: Simula Begin. Studentlitteratur (Lund, Sweden), Bratt Institut fuer neues Lernen (Goch, FRG), Chartwell-Bratt Ltd (Kent, England) (1979)
2. Chakravarty, M.M.T., Keller, G., Jones, S.L.P., Marlow, S.: Associated types with class. In: Palsberg, J., Abadi, M. (eds.) POPL, pp. 1–13. ACM Press, New York (2005)

3. Coquand, T., Pollack, R., Takeyama, M.: A logical framework with dependently typed records. In: Hofmann, M.O. (ed.) TLCA 2003. LNCS, vol. 2701, pp. 105–119. Springer, Heidelberg (2003)
4. Damas, L., Milner, R.: Principal type schemes for functional programs. In: POPL, Albuquerque, New, Mexico, pp. 207–212 (1982)
5. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: ICFP 2002, pp. 235–246. ACM Press, New York (2002)
6. Haftmann, F., Wenzel, M.: Constructive Type Classes in Isabelle. In: Altenkirch, T., McBride, C. (eds.) TYPES 2006. LNCS, vol. 4502, pp. 160–174. Springer, Heidelberg (2007)
7. Huffman, B., Matthews, J., White, P.: Axiomatic Constructor Classes in Isabelle/HOLCF. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 147–162. Springer, Heidelberg (2005)
8. Jones, M.P.: Type classes with functional dependencies. In: Smolka, G. (ed.) ESOP 2000 and ETAPS 2000. LNCS, vol. 1782, pp. 230–244. Springer, Heidelberg (2000)
9. Kahl, W., Scheffczyk, J.: Named instances for haskell type classes. In: Hinze, R. (ed.) A Comparative Study of Very Large Data Bases. LNCS, vol. 59. Springer, Heidelberg (2001)
10. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales - A Sectioning Concept for Isabelle. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 149–166. Springer, Heidelberg (1999)
11. Letouzey, P.: Programmation fonctionnelle certifie – L'extraction de programmes dans l'assistant Coq. PhD thesis, Universit Paris-Sud (July 2004)
12. Moors, A., Piessens, F., Odersky, M.: Generics of a higher kind. In: ECOOP 2008 (submitted, 2008)
13. Pollack, R.: Dependently typed records for representing mathematical structure. In: Aagaard, M.D., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869, pp. 462–479. Springer, Heidelberg (2000)
14. Saïbi, A.: Typing algorithm in type theory with inheritance. In: POPL, La Sorbonne, Paris, France, January 15-17, 1997, pp. 292–301. ACM Press, New York (1997)
15. Sozeau, M.: Subset coercions in Coq. In: Altenkirch, T., McBride, C. (eds.) TYPES 2006. LNCS, vol. 4502, pp. 237–252. Springer, Heidelberg (2007)
16. The Coq Development Team. The Coq Proof Assistant Reference Manual – Version V8.1 (July 2006), `http://coq.inria.fr`
17. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: POPL, Austin, Texas, pp. 60–76 (1989)
18. Wenzel, M.: Type classes and overloading in higher-order logic. In: Gunter, E.L., Felty, A.P. (eds.) TPHOLs 1997. LNCS, vol. 1275, pp. 307–322. Springer, Heidelberg (1997)
19. Wenzel, M., Paulson, L.: Isabelle/isar. In: Wiedijk, F. (ed.) The Seventeen Provers of the World. LNCS (LNAI), vol. 3600, pp. 41–49. Springer, Heidelberg (2006)