Typing Algorithm in Type Theory with Inheritance

Amokrane Saïbi
Amokrane.Saibi@inria.fr
INRIA Rocquencourt B.P. 105 - 78153 Le Chesnay CEDEX
Université Pierre et Marie Curie – Paris 75252

Abstract

We propose and study a new typing algorithm for dependent type theory. This new algorithm typechecks more terms by using inheritance between classes. This inheritance mechanism turns out to be powerful: it supports multiple inheritance, classes with parameters and uses new abstract classes **FUNCLASS** and **SORTCLASS** (respectively class of functions and sorts). We also defines classes as records, particularly suitable for the formal development of mathematical theories. This mechanism, implemented in the proof checker Coq, can be adapted to all typed λ-calculus.

1 Introduction

In the last years, proof checkers based on type theory appeared as convincing systems to formalize mathematics (especially constructive mathematics) and to prove correctness of software and hardware. In a proof checker, one can interactively build definitions, statements and proofs. The system is then able to check automatically whether the definitions are well-formed and the proofs are correct.

Modern systems are Coq[8] and LEGO[12], based on variants of the Calculus of Constructions, and NuPRL[10] and ALF based on variants of Martin-Löf's type theory. In all these theories, proofs are represented via the Curry-Howard isomorphism: each proposition corresponds to a type, and a proof of a proposition is an object inhabiting its corresponding type. Some proof checkers, like Coq and NuPRL, are also programming languages. The program is automatically extracted from the proof of its specification, and a metatheorical result [14] insures that the extracted program obeys its specification.

However there are two obstacles to the application of proof-checkers to large-scale formal developments. The first one is that the proof activity is tedious because one has to justify each reasoning step. This obstacle is (partially) solved using tools like decision procedures to automate certain parts of the proofs. The second inadequacy concerns the rigid grammar of proof checkers, very far from the mathematician usage. Thus the proof checkers are extended with *syntactic facilities* to improve their syntax. Among them, there is the *implicit argument synthesis*[17] to infer some polymorphic instantiations. For example, we write $(Comp\ f\ g)$ for the composition of $f : A \to B$ and $g : B \to C$, instead of $(Comp\ A\ B\ C\ f\ g)$ because we can deduce the first three arguments from the type of f and g. The definition mechanism is also a syntactic facility since we can use a name instead of its corresponding term. In the previous example $Comp$ is an abbreviation of $\lambda A : Set.\lambda B : Set.\lambda C : Set.\lambda f : A \to B.\lambda g : B \to C.\lambda a : A.(g\ (f\ a))$.

In [1] and [2], P. Aczel proposed a new syntactic facility based on inheritance to mimick some *abus de notations* used in mathematics. By inheritance, he refers to the possibility of applying a function $f : A \to B$ to a term $a : A'$ because A' can be seen in some sense as a subtype of A. For instance a group can be used as a set, or a monoid or any sub-structure of group, according to the context. P. Aczel proposed a notion of class and method where each method is associated to only one class but is inherited by its subclasses. The inheritance relation is then defined using functions called coercions: a class C inherits from a class D if there is a coercion from the type of C to the type of D. This work was extended by G. Barthe [5] to the multiple inheritance case. In this work, each function is a method, and thus does not have to be declared. However there is no implementation of these inheritance mechanisms.

POPL 97, Paris, France

As a first step and in collaboration with A. Bailey, the author defined and implemented in LEGO, an inheritance mechanism with multiple inheritance and classes with parameters (not available in previous systems). In this paper, I present an improved mechanism where I introduced two abstract classes **FUNCLASS** and **SORTCLASS** to capture more abuses of notation. For instance:

- we allow to write $x : A$ when A is not a type, but can be seen in a certain sense as a type: set, group, category etc.
- we allow to write $(f\ a)$ when f is not a function, but can be seen in a certain sense as a function: bijection, functor, any structure morphism etc.

We present our mechanism for any typed λ-calculus with dependent types (or PTS[4, 15]: Pure Type System) with constants and decidable typing. To allow these abuses of notations, we do not modify the theory of PTSs but only modify the typing algorithm to take account of inheritance. We proved the correction of this new typing algorithm.

For a PTS with inductive types, we defined *structures with inheritance* (classes as records), and sketched the definition of *class operators*. This inheritance mechanism, implemented in Coq, is heavily used in the formalisation of category theory[19].

The paper is organised as follows: in the next section, we give the typing algorithm for a PTS with constants. In section 3, we show how to build the inheritance graph. In section 4, we present the typing algorithm with inheritance. In section 5, we describe the structures with inheritance and class operators. In section 6, we exemplify our inheritance mechanism in the formalisation of a fragment of abstract algebra. Finally, we discuss related work.

2 Type Theory

2.1 PTS

Let (S, Ax, R) be a PTS with global constants where S is a set of sorts, $Ax \subseteq S \times S$ a set of axioms and $R \subseteq S \times S \times S$ the set of allowed products. The pseudo-terms set T is given by the following grammar:

T	$::=$	c	constant
	$\mid$	x	variable
	$\mid$	$(x : T)T$	dependent type $\forall x : T.T$
	$\mid$	$[x : T]T$	abstraction $\lambda x : T.T$
	$\mid$	$(T\ T)$	functional application
	$\mid$	s	sort

In $[x : T]U$ and $(x : T)U$, the first occurrence of x binds the free occurrences of x in U but not in T. The construction $(x : T)U$ corresponds to the dependent type of functions, where the result type of the function depends on the value of its argument. When x does not appear in U, $(x : T)U$ is the usual functional type $T \to U$. The deduction rules of this theory derive judgements $\Gamma \vdash t : T$ with the meaning "T is a type of t in the context Γ". A context is a list of bindings of the form $x : T$ and $c := t :: T$, it associates a type (here T) to variables and a type and a value (resp. T and t) to constants. We refer to the type of a name (constant or variable) x in a context Γ by Γ_x, and to the value of c by $val(c, \Gamma)$. We have first to define the rules of well-formedness of contexts, noted $\Gamma \vdash$. The rule (CTXT) concerns the empty context $\emptyset$ and (VAR) and (CONST) declare resp. new variables and constants.

$$(\text{CTXT})\ \frac{}{\emptyset \vdash}$$

$$(\text{VAR})\ \frac{\Gamma \vdash T : s \qquad s \in S \qquad x \notin \Gamma}{\Gamma; x : T \vdash}$$

$$(\text{CONST})\ \frac{\Gamma \vdash t : T \qquad c \notin \Gamma}{\Gamma; c := t :: T \vdash}$$

Finally, the deduction rules are:

$$(\text{AXIOM})\ \frac{(s_1, s_2) \in Ax}{\Gamma \vdash s_1 : s_2}$$

$$(\text{NAME})\ \frac{x \in \Gamma}{\Gamma \vdash x : \Gamma_x}$$

$$(\text{LAM})\ \frac{\Gamma; x : T \vdash t : U \qquad \Gamma \vdash (x : T)U : s}{\Gamma \vdash [x : T]t : (x : T)U}$$

$$(\text{PROD})\ \frac{\Gamma \vdash T : s_1 \qquad \Gamma; x : T \vdash U : s_2 \qquad (s_1, s_2, s_3) \in R}{\Gamma \vdash (x : T)U : s_3}$$

$$(\text{APP})\ \frac{\Gamma \vdash t : (x : T)U \qquad \Gamma \vdash u : T}{\Gamma \vdash (t\ u) : U[x \leftarrow u]}$$

$$(\text{CONV})\ \frac{\Gamma \vdash T : s \qquad \Gamma \vdash t : U \qquad T =_{\beta\delta} U}{\Gamma \vdash t : T}$$

We say that a term is a *type* if it is well-typed and its type is a sort.

The $=_{\beta\delta}$ conversion is the equality generated by the reduction rules:

β $([x:T]t\ u) \rightarrow_\beta t[x \leftarrow u]$
δ $c \rightarrow_\delta val(c,\Gamma)$

The rule β corresponds to the replacement of the formal parameter of a function by its effective argument. The term $t[x \leftarrow u]$ denotes the term t with its free occurrences of x substitued by u. As to δ, it is the replacement of a constant by its value. Remark that in the substitution process, we do not have to unfold constants.

In [15] is given a more complete presentation of PTSs with local and global constants.

2.2 Typing Algorithm

We now describe the typing algorithm for terms, based on the previous rules. Let Γ be a context, t be a term, we infer a type T such that $\Gamma \vdash t : T$. We will present also this algorithm using inference rules where the shape of judgements is $\Gamma \vdash_I t : T$, and with the meaning 'T is the infered type for T in Γ". It consists principally in eliminating the indeterminism introduced by the rule (CONV). The inferred type is said to be *principal* because only the "necessary" δ-reductions are performed. To guarantee the unicity of typing, we should require more hypotheses on our theory. The PTS should be functional i.e. $R_{s_1,s_2} = \{s_3 | (s_1, s_2, s_3) \in R\}$ may contain at most one sort, which will be noted R_{s_1,s_2} when it exists. We make the same hypotheses for $Ax_s = \{s' | (s, s') \in A\}$. Finally we suppose that $\rightarrow_{\beta\delta}$ is confluent and strongly normalising, thus $=_{\beta\delta}$ is decidable.

Notations.

- s eventually with subscribe (s_1, s_2...) represent sorts.
- In the notation $t \equiv u$, u matches t and care is taken about names of bound variables. For example, in $(f : bool \rightarrow nat)(b : bool)(vect\ (f\ b)) \equiv (x : A)B$, x represents f, A represents $bool \rightarrow nat$ and B represents $(b : bool)(vect\ (x\ b))$.
- We will often note $(...((f\ a_1)\ a_2)...a_n)$ by $(f\ a_1...a_n)$.

The rules of the typing algorithm are:

$$(\text{CTXT})\ \frac{}{\emptyset \vdash_I}$$

$$(\text{VAR})\ \frac{\Gamma \vdash_I T : U \qquad \overline{U} \equiv s \qquad x \notin \Gamma}{\Gamma; x : T \vdash_I}$$

$$(\text{CONST})\ \frac{\Gamma \vdash_I t : T' \qquad \Gamma \vdash_I T : U \qquad T =_{\beta\delta} T' \qquad c \notin \Gamma}{\Gamma; c := t :: T \vdash_I}$$

$$(\text{AXIOM})\ \frac{}{\Gamma \vdash_I s : Ax_s}$$

$$(\text{NAME})\ \frac{x \in \Gamma}{\Gamma \vdash_I x : \Gamma_x}$$

$$(\text{LAM})\ \frac{\Gamma; x : T \vdash_I t : U \qquad \Gamma \vdash_I (x : T)U : s}{\Gamma \vdash_I [x : T]t : (x : T)U}$$

$$(\text{PROD})\ \frac{\Gamma \vdash_I T : T' \qquad \overline{T'} \equiv s_1 \qquad \Gamma; x : T \vdash_I U : U' \qquad \overline{U'} \equiv s_2}{\Gamma \vdash_I (x : T)U : R_{s_1,s_2}}$$

$$(\text{APP})\ \frac{\Gamma \vdash_I t : T \qquad \overline{T} \equiv (x : T_1)T_2 \qquad \Gamma \vdash_I u : U \qquad T_1 =_{\beta\delta} U}{\Gamma \vdash_I (t\ u) : T_2[x \leftarrow u]}$$

The term $\bar{t}$ (t is a well typed term) is the head normal form of t. Remark that the δ-reductions are performed only when necessary, and that no reduction is performed when the shape of the term is determined.

$$\begin{array}{lcl}
\overline{s} & = & s \\
\overline{c} & = & \overline{val(c,\Gamma)} \\
\overline{x} & = & x \\
\overline{(x : T)U} & = & (x : T)U \\
\overline{[x : T]t} & = & [x : T]t \\
\overline{([x : T]t\ u_1...u_n)} & = & \overline{(t[x \leftarrow u_1]\ u_2...u_n)} \\
\overline{(x\ u_1...u_n)} & = & (x\ u_1...u_n) \\
\overline{(t\ u_1...u_n)} & = & (\bar{t}\ u_1...u_n) \text{ if } t \text{ is neither an abstraction, nor a variable}
\end{array}$$

In the declaration of a constant, its type is not essential since we can infer it from its value. However it permits to impose a principal type to the constant. We generalize this control to all terms, by introducing *terms with cast* of the form $t :: T$, where T should be the principal type of t. The inference rule of this construction is:

$$(\text{CAST})\ \frac{\Gamma \vdash_I t : T' \qquad \Gamma \vdash_I T : U \qquad T =_{\beta\delta} T'}{\Gamma \vdash_I t :: T : T}$$

The algorithmic presentation of PTSs is discussed in detail in [16].

Remark. For simplicity and generality, we only presented the PTS fragment of Coq with constants but without inductive types and universes. However they are considered in our implementation: universes are treated as simple sorts and inductive types and their constructors as constants. The typing algorithm in Coq is more complex than the one we presented because it synthesises implicit arguments and manage floating universes[9].

2.3 Subtyping

Let $\leq$ be a preorder on types. We obtain a theory with subtyping if we add the subsumption (SUB1) rule.

$$(\text{SUB1})\ \frac{\Gamma \vdash t : T \qquad T \leq T'}{\Gamma \vdash t : T'}$$

To keep the theory strongly typed in the sense that every term has at most one type, we replace (SUB1) by (SUB2) where $c_{T,T'}$ is the unique coercion between T and T'.

$$(\text{SUB2})\ \frac{\Gamma \vdash t : T \qquad c_{T,T'} : T \leq T'}{\Gamma \vdash (c_{T,T'}\ t) : T'}$$

In this system, we are not interested in adding any expressive power to our theory , but only convenience. Given a term, possibly not typable, we are interested in the problem of determining if it can be well typed modulo insertion of appropriate coercions. Such a typing algorithm is given in [3] in the context of simply typed λ-calculus with recursive types. A weaker rule is:

$$(\text{APP'})\ \frac{\Gamma \vdash f : (x : U)V \qquad \Gamma \vdash t : T \qquad c_{T,U} : T \leq U}{\Gamma \vdash (f\ (c_{T,U}\ t)) : V[x \leftarrow (c_{T,U}\ t)]}$$

The solution we will describe is closer to the last one. To build the relation $\leq$, we consider a more disciplined form of subtyping based on inheritance.

3 Inheritance Graph

3.1 Classes

A class with n parameters is any name C with a principal type $(x_1 : A_1)...(x_n : A_n)s$. Thus a class with parameters is considered as a single class and not as a family of classes. In addition to these classes, we have `SORTCLASS` and `FUNCLASS` two abstract classes of sorts and functions, respectively. To determine the class of a term, one has to apply the function Cl to its principal type.

$Cl(s)$	=	`SORTCLASS`
$Cl((x : A)B)$	=	`FUNCLASS`
$Cl((C\ t_1...t_n))$	=	C if C is a class with n parameters

The function Cl is undefined otherwise. Remark that for efficiency reasons, we use syntactic equality between classes and not convertibility (like in [5]). Thus two classes with different names will always be considered different even when they have the same value.

3.2 Coercions

A name f can be declared as coercion between a source class C and a target class D if its principal type is of the form $(x_1 : A_1)...(x_n : A_n)(y : (C\ x_1...x_n))(D\ u_1...u_m)$ where n and m are the number of parameters of respectively C and D. We then write $f : C \rightarrowtail D$. The abstract classes `FUNCLASS` and `SORTCLASS` cannot be source classes.

The restriction on the type of coercions, called *the uniform inheritance condition*, ensures that any coercion can be applied to any object of its source class. Indeed let $f : C \rightarrowtail D$ be a coercion and $t : (C\ t_1...t_n)$ be an object of C, we define the application of f to t as $f@t = (f\ t_1...t_n\ t)$. Its type is $f\{t\} = (D\ u_1^*...u_m^*)$ where $u_i^* = u_i[y \leftarrow t][x_n \leftarrow t_n]...[x_1 \leftarrow t_1]$. Without the uniform inheritance condition i.e. with a coercion $f : (x_1 : T_1)...(x_k : T_k)(y : (C\ u_1...u_n))(D\ v_1...v_m)$, one can only define $f@t$ as $(f\ ?...?\ t)$, and then unknown parameters "?" have to be synthetised by an appropriate algorithm. In section 3.4 we will see how to go around this condition using *identity coercions*.

3.3 Building the Inheritance Graph

Coercions form an inheritance graph with classes as nodes. A class C is said to be a subclass of D if there is a coercion path in the graph from C to D; we also say that C inherits from D. Our mechanism supports multiple inheritance since a class may inherit from several classes, contrary to simple inheritance where a class inherits from at most one class. However there must be at most one path between two classes. If this is not the case, only the oldest one is *valid* and the others are ignored. This choice is discussed in section 7.

Notations.

- We represent a coercion path by the ordered list of coercions composing it, for instance $[f_1; ...; f_k]$.
- $p : C \rightarrowtail D$ means that p is a coercion path between C and D.
- $\Delta + f$ is the graph obtained by adding the coercion f to the graph Δ.
- Δ_C^D is the path coercion from C to D in the graph Δ. Its value is noted $\perp$ if the classes C and D are not connected.
- $p \hat{} q$ is the concatenation of the coercion paths p and q.

We extend notations for coercions to path coercions. Let $[f_1; ...; f_k] : C \rightarrowtail D$ be a coercion path:

- $[f_1; ...; f_k]@t = f_k@(...(f_1@t))$

• $[f_1;...;f_k]t = f_k\{...f_1\{t\}\}$

It is possible to have path coercions between the same source and target class. We can also have two opposite path coercions $p : C \longmapsto D$ and $q : D \longmapsto C$ linking equivalent representations of a same mathematical object.

We explain now how to compute $\Delta + f$. We represent an inheritance graph as the set of its valid coercion paths. Let $f : C \longmapsto D$ be a coercion to add to the graph Δ. We compute first all the new generated coercion paths.

$$\begin{aligned} New(\Delta, f) = \; & \{p\hat{}[t] \mid (p : B \longmapsto C) \in \Delta\} \cup \\ & \{[t]\hat{}q \mid (q : D \longmapsto E) \in \Delta\} \cup \\ & \{p\hat{}[t]\hat{}q \mid (p : B \longmapsto C) \in \Delta \wedge \\ & \qquad (q : D \longmapsto E) \in \Delta\} \end{aligned}$$

Notice that if Δ contains only valid coercion paths, then it is also the case for $New(\Delta, f)$. The graph $\Delta + f$ is defined as:

$$\Delta \cup \{p \mid (p : X \longmapsto Y) \in New(\Delta, f) \wedge \Delta_X^Y = \bot\}$$

3.4 Identity Coercions

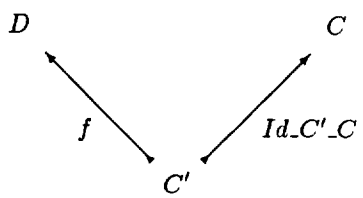
Let C and D be two classes with respectively n and m parameters and $f : (x_1 : T_1)...(x_k : T_k)(y : (C\ u_1....u_n))\ (D\ v_1....v_m)$ a function which does not verify the uniform inheritance condition. To declare f as coercion, one has first to declare a subclass C' of C:

$$C' := [x_1 : T_1]...[x_k : T_k](C\ u_1....u_n)$$

We then define an *identity coercion* between C' and C:

$$\begin{aligned} Id\_C'\_C := & [x_1 : T_1]...[x_k : T_k][y : (C'\ x_1....x_k)] \\ & (y :: (C\ u_1....u_n)) \end{aligned}$$

We can now declare f as coercion from C' to D, since we can "cast" its type as $(x_1 : T_1)...(x_k : T_k)(y : (C'\ x_1....x_k))(D\ v_1....v_m)$.



We choose to give a special status to identity coercions by redefining $Id\_C'\_C@t = t$ where $t : (C'\ t_1...t_k)$. The application of an identity coercion to a term only changes its type – the definition of $Id\_C'\_C\{t\}$ does not change.

4 Typing Algorithm with Inheritance

We propose a new typing algorithm based on our inheritance mechanism. The new algorithm typechecks strictly more terms than $\vdash_I$ by using inheritance. Let t be a term, Γ be a context and Δ be an inheritance graph, the algorithm returns terms t' and T such that $\Gamma \vdash t' : T$. The term t' (called the explicit term) is obtained by insertion of coercions in t (called the implicit term). We will use the notation $\Delta, \Gamma \vdash t \Rightarrow t' : T$.

We now define functions that transform a term t of type T to a required form (a sort, a function or a specified type) by insertion of coercions. In the first two cases, we return a pair consisting of the transformed term and its type (a sort or a functional type). In the last case, only the transformed term is returned. The inheritance is only used if the term t is not already in the required form. In each definition, the cases order is important. The function fails if no case matches.

Notation. $\langle t, u\rangle$ is the pair of the terms t and u, and x^1 and x^2 are respectively the first and second components of the pair x.

Let Δ be an inheritance graph.

• Inheritance towards a sort.

$$\Delta^{\mathtt{SORT}}(t : T) = \begin{cases} \langle t, \overline{T}\rangle & \text{if } \overline{T} \equiv s \\ \langle p@t, p\{t\}\rangle & \text{with } p = \Delta^{\mathtt{SORTCLASS}}_{Cl(T)} \end{cases}$$

• Inheritance towards a function.

$$\Delta^{\mathtt{FUN}}(t : T) = \begin{cases} \langle t, \overline{T}\rangle & \text{if } \overline{T} \equiv (x : A)B \\ \langle p@t, p\{t\}\rangle & \text{with } p = \Delta^{\mathtt{FUNCLASS}}_{Cl(T)} \end{cases}$$

• Inheritance towards a type.

$$\Delta(t : T, U) = \begin{cases} t & \text{if } T =_{\beta\delta} U \\ \Delta^{Cl(U)}_{Cl(T)}@t & \text{if } \Delta^{Cl(U)}_{Cl(T)}\{t\} =_{\beta\delta} U \end{cases}$$

$$\Delta(t : (y : T_1)T_2, (z : U_1)U_2) = [x : T_1']\Delta(t' : T_2^*, U_2^*)$$

with $t \equiv [x : T_1']t'$, $T_1 =_{\beta\delta} T_1'$, $T_1 =_{\beta\delta} U_1$,
$T_2^* = T_2[y \leftarrow x]$ and $U_2^* = U_2[z \leftarrow x]$

$$\Delta(t : (y : T_1)T_2, (z : U_1)U_2) = [x : U_1]\Delta((t\ u) : T_2^*, U_2^*)$$

with x is a new variable, $u = \Delta(x : U_1, T_1)$,
$T_2^* = T_2[y \leftarrow u]$ and $U_2^* = U_2[z \leftarrow x]$

The last case is the dependent version of the anti-monotonic subtyping rule of the function spaces.

$$\frac{f : T_1 \to T_2 \qquad c_{U_1,T_1} : U_1 \leq T1 \qquad c_{T_2,U_2} : T_2 \leq U_2}{[x : U_1](c_{T_2,U_2}\ (f\ (c_{U_1,T_1}\ x))) : U_1 \to U_2}$$

The last but first case is a particular case (optimized) of the last one, but avoids superfluous abstractions when t is an abstraction.

We present only the modified rules of the typing algorithm. The terms (types and values) are in explicit form in the context; when a name is declared in a given context, its type and value are first explicitated in this context and then it is declared using (VAR) or (CONST). The extension of the graph by a coercion is made by the rule (COE), "f is a coercion" means that f verifies the conditions of 3.2.

$$(\text{COE})\ \frac{f \in \Gamma \qquad f \text{ is a coercion}}{\Delta + f, \Gamma \vdash}$$

We use a new form of judgement $\Delta, \Gamma \vdash_S A \Rightarrow A' : s$, with the meaning "the term A is transformed to the term A' which is a type".

$$(\text{S})\ \frac{\Delta, \Gamma \vdash A \Rightarrow A' : T}{\Delta, \Gamma \vdash_S A \Rightarrow \Delta^{\mathsf{SORT}}(A' : T)^1 : \Delta^{\mathsf{SORT}}(A' : T)^2}$$

This judgement is used in the following rules.

$$(\text{PROD})\ \frac{\Delta, \Gamma \vdash_S T \Rightarrow T' : s_1 \qquad \Delta, \Gamma; x : T' \vdash_S U \Rightarrow U' : s_2}{\Delta, \Gamma \vdash (x : T)U \Rightarrow (x : T')U' : R_{s_1,s_2}}$$

$$(\text{LAM})\ \frac{\Delta, \Gamma \vdash_S T \Rightarrow T' : s \qquad \Delta, \Gamma; x : T' \vdash t \Rightarrow t' : U \qquad \Delta, \Gamma \vdash_S (x : T')U \Rightarrow V : s}{\Delta, \Gamma \vdash [x : T]t \Rightarrow [x : A']t' : V}$$

In the (APP) rule, the inheritance coercion is used in both function and argument positions.

$$(\text{APP})\ \frac{\Delta, \Gamma \vdash t \Rightarrow t' : T \qquad \Delta, \Gamma \vdash u \Rightarrow u' : U \qquad \Delta^{\mathsf{FUN}}(t' : T)^2 \equiv (x : A)B}{\Delta, \Gamma \vdash (t\ u) \Rightarrow (\Delta^{\mathsf{FUN}}(t' : T)^1\ \Delta(u' : U, A)) : B[x \leftarrow \Delta(u' : U, A)]}$$

In the typing of a cast $t :: T$, T must be a type and be convertible with the computed type of t modulo inheritance.

$$(\text{CAST})\ \frac{\Delta, \Gamma \vdash t \Rightarrow t' : T' \qquad \Delta, \Gamma \vdash_S T \Rightarrow T'' : s}{\Delta, \Gamma \vdash t :: T \Rightarrow \Delta(t' : T', T'') :: T'' : T''}$$

The algorithm verifies the following properties:

- It is correct, in the sens that if $\Delta, \Gamma \vdash t \Rightarrow t' : T'$ then t' is a well typed term and T' is its inferred type, i.e. $\Gamma \vdash_I t' : T'$.
- It is a conservative extension of $\vdash_I$. If $\Gamma \vdash_I t : T$ then $\Delta, \Gamma \vdash t \Rightarrow t : T$ for all Δ. Thus our new algorithm behaves like $\vdash_I$ for the well typed terms. In particular they give exactly the same answers when Δ is the empty graph.

The proofs of these properties are given in detail in [18].

5 Extensions

We propose two extensions of our mechanism in the context of PTSs with inductive types[13]. The first one is the definition of *Structures with Inheritance* (or classes as records), useful to represent inheritance between mathematical theories expressed as records. The second one is about inheritance between parametrised inductive types, or *class operators*. This last extension is not implemented, thus we kept it for further research.

We recall that an inductive type is specified by giving its name and type and the names and types of its constructors. For instance, the type `nat` of natural numbers is an inductive type with two constructors `O` and `S`.

```
Inductive Definition nat : Set := O : nat
                                | S : nat -> nat.
```

We can also define inductive types with parameters, like `list` which is an inductive type with one parameter `A`, and two constructors: `nil` for the empty list and `cons` to add an element in front of a list.

```
Inductive List [A:Set] : Set :=
  nil  : (List A)
| cons : A -> (List A) -> (List A).
```

From an inductive type definition, an elimination principle is generated, corresponding both to the construction of a function following a primitive recursive definition scheme and to proofs by inductions. See [13] for more details.

5.1 Structures with Inheritance

Usually in programming languages, classes are records, and one defines a subclass C of an existing class D by extending it with new fields. So there exists implicitely a coercion (forgetful or inclusion function) from C to D: it simply forgets the supplementary fields of C. In a proof checker with inductive types, records can be obtained as special case of inductive types. In particuler, in Coq a record R with m parameters and n fields is

declared by:

$Structure\ R\ [p_1 : u_1; ....; p_m : u_m] : s :=$
$R\_Cons\ \{L_1 : t_1; ....; L_n : t_n\}$

The name of the unique constructor R_Cons is optional and may be omitted. Every field has a label L_i and a type t_i. Of course all labels must be different. In a dependent type theory context, the type of a label may depends on its predecessor labels. This record is then translated into a inductive type R with one constructor R_Cons (if R_Cons is not supplied, $Build\_R$ is used).

$Inductive\ R\ [p_1 : u_1; ....; p_m : u_m] : s :=$
$R\_Cons : (L_1 : t_1)....(L_n : t_n)(R\ p_1 ... p_m)$

Projection functions are also automatically generated[1], using pattern-matching associated to inductive types.

$L_i := [p_1 : u_1; ....; p_m : u_m][x : (R\ p_1 ... p_m)]$
$Case\ x\ of\ (R\_Cons\ p_1 ... p_n\ x_1 ... x_n) => x_i$

The term $(R\_Cons\ a_1 ... a_m\ u_1 ... u_n)$ is the object of type $(R\ a_1 ... a_m)$ built from the constituents $u_1 ... u_n$. And finally, we obtain as special case of the ι inductive types reduction:

$$(L_i\ a_1 ... a_n\ (R\_Cons\ a_1 ... a_m\ u_1 ... u_n)) \rightarrow_\iota u_i$$

A structure with inheritance is a record where certain projection functions are coercions. Its syntax is:

$Structure\ R\ [p_1 : u_1; ....; p_m : u_m] : s :=$
$R\_Cons\ \{L_1\ [:\ or\ :>]\ t_1; ....; L_n\ [:\ or\ :>]\ t_n\}$

where $[:\ or\ :>]$ is : or :>. If $L_i :> t_i$, then L_i is declared as coercion from R to $Cl(t_i)$ (if $Cl(t_i)$ exists). Remark that L_i always verifies the uniform inheritance condition.

In Lego[12], records correspond to Σ-types (i.e. dependent product types) but without names for the fields. As to ALF, [20] extends its theory with labelled records.

5.2 Class Operators

Our inheritance mechanism does not handle data types. For instance it cannot transform a list of natural numbers to a list of rational numbers, even if there is a coercion between natural and rational numbers. The situation is the same for the other data types like products, vectors etc. All these data types are special kinds of parametrised inductive types. To each parametrised inductive type corresponds a subtyping rule. For instance:

$$\frac{A \leq B}{(List\ A) \leq (List\ B)}$$

$$\frac{A \leq B \qquad A' \leq B'}{A * A' \leq B * B'}$$

In the case of lists, the coercion between $(List\ A)$ and $(List\ B)$ is recursively defined for any coercion f between A and B.

```
Recursive Definition Iter_List [A, B:Set; f:A -> B] :
                                  (List A) -> (List B) :=
   (nil A)          => (nil B)
 | (cons A a rest)  => (cons B (f a)
                                  (Iter_List A B f rest)).
```

Such a coercion can be automatically defined for any parametrised inductive type on the basis of its constructor types. We then extend the transformation function $\Delta(t : T, U)$ by a new case. For the lists, this case is:

$$\Delta(t : (List\ T), (List\ U)) = (Iter\_List\ T\ U\ [x : T]\Delta(x : T, U)\ t)$$

6 Example

We use our inheritance mechanism in the development of a small fragment of abstract algebra in Coq.

Notations. We recall some Coq notations.

- `Definition c := t.` defines a constant c with value t. Its type is automatically computed.
- `Class` C. declares C as a class.
- `Coercion` f : C `>->` D . declares f as a coercion between C and D.
- `Identity Coercion` f : C `>->` D . declares and defines f as a local coercion between C and D.
- `==` is the Leibniz equality.
- `/\` is the conjunction connective.

We begin by defining the `Monoid` type of monoids as a record consisting of a carrier, a binary operation on this carrier and a distinguished element of the carrier. These constituents must verifie the three monoid laws `Ass`, `Idl` and `Idr`. These laws form a class `IsMonoid` with three parameters.

```
Definition IsMonoid := [A:Type][op:A->A->A][i:A]
          (Ass A op)/\(Idl A op i)/\(Idr A op i).

Class IsMonoid.
```

[1] Some of them cannot be defined, following the sort s and the type t_i. See chapter 6 of [8] for details

As to the type Monoid, it is a class inheriting from SORTCLASS and IsMonoid.

```
Structure Monoid : Type :=
 {Carrier :> Type;
  Op      :  Carrier -> Carrier -> Carrier;
  I       :  Carrier;
  Mprop   :> (IsMonoid Carrier Op I)}.
```

Remark the constructor Build_Monoid has the type:

```
(Carrier:Type)(Op:Carrier->Carrier->Carrier)(I:Carrier)
(IsMonoid Carrier Op I)->Monoid
```

We can then declare it as a coercion from IsMonoid to Monoid.

```
Coercion Build_Monoid : IsMonoid >-> Monoid.
```

We build the class Group of groups on the class of monoids by adding fields for the inverse function and the group laws.

```
Definition Inv_l := [M:Monoid][f:M->M](a:M)
                    (Op a (f a)) == (I M).

Definition Inv_r := [M:Monoid][f:M->M](a:M)
                    (Op (f a) a) == (I M).

Structure Group : Type :=
 {G_M   :> Monoid;
  Ginv  :> G_M -> G_M;
  Gprop :  (Inv_l G_M Ginv) /\ (Inv_r G_M Ginv)}.
```

We define the commutativity property to be able to build the class AbGroup of abelian groups.

```
Definition Comm := [G:Group](a,b:G) (Op a b) == (Op b a).

Structure AbGroup : Type :=
 {AbG_G   :> Group;
  AbGprop : (Comm AbG_G)}.
```

As test, we give the statement "$(b^{-1_G} + a) + (b + a^{-1_G}) = 1_G$" where G is an abelian group.

```
(G:AbGroup)(a,b:G)
     (Op (Op (G b) a) (Op b (G a))) == (I G).
```

Consider now the class MorMonoid of monoid morphisms parametrised by two monoids. It consists of a function between the monoid carrtiers, preserving the monoid structure.

```
Definition Pres_Mon := [A,B:Monoid][f:A->B]
          (f (I A)) == (I B) /\
          (a,b:A)(f (Op a b)) == (Op (f a) (f b)).

Structure MorMonoid [A,B:Monoid] : Type :=
 {ApM       :> A -> B;
  MorM_pres :  (Pres_Mon A B ApM)}.
```

We extend this class to define the class MorGroup of group morphisms.

```
Definition Pres_Grp := [A,B:Group][f:(MorMonoid A B)]
                       (a:A)(f (A a)) == (B (f a)).

Structure MorGroup [A,B:Group] : Type :=
 {MG_MM     :> (MorMonoid A B);
  MorG_pres :  (Pres_Grp A B MG_MM)}.
```

The statement "$f(a^{-1_A} + 1_A) = (f\ a)^{-1_B}$" can then be written as follows:

```
(A,B:AbGroup)(f:(MorGroup A B))(a:A)
  (f (Op (A a) (I A))) == (B (f a))
```

The equivalent statement without inheritance is the cumbersome formula:

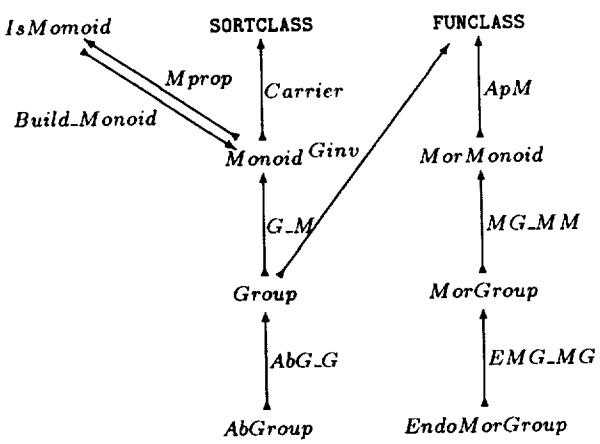
```
(A,B:AbGroup)(f:(MorGroup (AbG_G A) (AbG_G B)))
  (a:(Carrier (G_M (AbG_G A))))
    (ApM (G_M (AbG_G A)) (G_M (AbG_G B))
         (MG_MM (AbG_G A) (AbG_G B) f)
         (Op (G_M (AbG_G A)) (Ginv (AbG_G A) a)
             (I (G_M (AbG_G A)))))
== (Ginv (AbG_G B) (ApM (G_M (AbG_G A)) (G_M (AbG_G B))
         (MG_MM (AbG_G A) (AbG_G B) f) a))
```

We end by definig the class of group endomorphisms as a subclass of group morphisms class. The identity coercion (see 3.4) is automatically built by the instruction Identity Coercion.

```
Definition EndoMorGroup := [G:Group](MorGroup G G).

Identity Coercion EMG_MG : EndoMorGroup >-> MorGroup.
```

The inheritance graph of the example is:



7 Comparison with Previous Work

P. Aczel's work on classes is the precursory in using inheritance as syntactic facility. He defined an inheritance mechanism using constants but only with classes

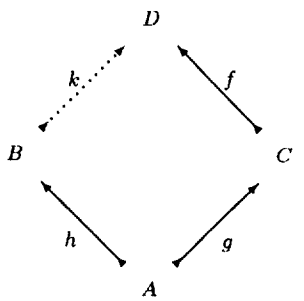
without parameters and simple inheritance. In [6], this mechanism is reformulated in the proof-checker ALF using records.

G. Barthe[5] defined another mechanism with multiple inheritance but without constants. Coercions are then defined between terms – the notion of class is not explicitly defined. This solution cannot be efficiently implemented because in order to determine the coercion between two types one should look for a coercion with a domain convertible with the first type and a codomain convertible with the second. It is the reason why we choose to consider classes as constants, and then tests between types become simple tests between names. He also adopted another solution for the multiple paths between classes; a coercion is added only if the obtained graph is *coherent*, i.e:

1. for any circular path $p : C \longmapsto C$, $(p\ x) =_\beta x$ where x is a fresh variable of type C.
2. for any two paths $p, q : C \longmapsto D$, $(p\ x) =_\beta (q\ x)$ where x is a fresh variable of type C.

The extension of this test to the classes with parameters case is easy. Let C be a class of type $(x_1 : A_1)...(x_n : A_n)s$, and $p, q : C \longmapsto D$ be two paths, we must verify $p@x =_{\beta\delta} q@x$ with $x : (C\ x_1...x_n)$ and $x_1 : A_1...x_n : A_n$ fresh variables. We do not use this coherence condition because it is too restrictive in practice, when we use inductively defined objects.

For instance in the below graph, the coercion $k : B \longmapsto D$ will be rejected if $[g; f]$ and $[h; k]$ are not convertible.



In our case, no test is made, $[k]$ is a valid path in the graph, but the path $[h; k]$ is ignored.

It is better to replace conversion by Leibniz equality "=" (generated by reflexivity axiom) to compare path coercions, because "=" is a bigger relation than conversion; for instance, we can prove $(x : nat)x + 0 = x$ by induction, even if x and $x + 0$ are not convertible. But in this case, the user has to prove equalities between multiple paths. Such a mechanism is still in study in Coq.

Another difference is that in [5] (and also [11]), the theory is extended by adding coercions related inference rules, and a new computation rule for insertion of coercions. They then study the metatheory of the resulting theory, but no typing algorithm is given. Luo [11] studies also the subtyping in type theory with inductive types.
In [7] is presented a polymorphic calculus for the representation of inheritance.

8 Conclusion

We presented an inheritance mechanism that we implemented in the proof checker Coq. The use of this mechanism, with other facilities as the implicit argument synthesis and infix notations, make mathematical statements more readable. We used it in [19]. We are thus closing the gap with standard mathematical practice, although some supplementary overloading mechanisms are obviously still lacking in order to implement the usual abuses of notation. We think about the overloading of function names. Indeed mathematicians can use a same symbol or name for different notions, as "=" or "+"; its meaning is then deduced from its context. In certain cases, inheritance can be seen as overloading. For instance, the symbol + (`Op` in section 6) is used as an operation on monoids, groups etc. On the other hand, several cases cannot be treated using inheritance, because we cannot always define coercions between types. For instance the equality between natural numbers and the equality between functors cannot have the same name because natural numbers class anf functors class cannot be connected by inheritance. In these cases, we need another overloading mechanism which choose the function to apply according to the type of its arguments. Such a mechanism is described in [2].

Acknowledgments

Thanks to Peter Aczel, Anthony Bailey and Gilles Barthe for useful discussions on inheritance in type theory.
Thanks to Christine Paulin-Mohring, Gérard Huet and Thérèse Hardin for their comments on an earlier version of this paper.
This work is partially supported by the ESPRIT project "TYPES: Types for Programs and Proofs". We would like to thank the referees for their efforts in improving the presentation of the paper.

References

[1] P. Aczel. *A notion of class for theory development in algebra (in a predicative type theory).* Presented at Workshop of Types for Proofs and Programs, Bastad, Sweden, June 1994.

[2] P. Aczel. *Simple overloading for type theories.* Privately circulated notes, 1994.

[3] R. Amadio, L. Cardelli. *Subtyping Recursive Types.* in ACM-TOPLAS, vol. 15, no 4, p. 575-631, 1993.

[4] H. P. Barendregt. *Typed Lambda calculi.* in Handbook of Logic in Computer Science, S. Abramsky et al. (Eds.), Oxford University Press, 117-309, 1992.

[5] G. Barthe. *Inheritance in type theory.* Draft paper, 1995.

[6] G. Betarte. *Classes and overloading in type theory with record types.* Draft paper, 1995.

[7] V. Breazu-Tannen, C. Gunter, T. Coquand, A. Scedrov. *Inheritance as implicit coercion.* Information and Computation, Vol 93, pp 172-221, 1991.

[8] C. Cornes, J. Courant, J-C. Filliatre, G. Huet, P. Manoury, C. Munoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, B. Werner. *The Coq Proof Assistant Reference Manual, version 5.10.* Rapport technique INRIA 0177, 1995.

[9] G. Huet. *Extending the calculus of constructions with Type:Type.* manuscrit non publié, Avril 1987.

[10] P. Jackson. *Enhancing the NuPRL proof development system and applying it to computationnal abstract algebra.* Phd thesis, Cornell University, 1995.

[11] Z. Luo. *Coercive Subtyping.* Draft, 1995.

[12] Z. Luo et R. Pollack. *LEGO proof development system: user's manual.* Technical Report ECS-LFCS-92-211, LFCS, Computer Science Dept. University of Edinburg, Mai 1992.

[13] C. Paulin-Mohring. *Inductive Definitions in the System Coq - Rules and Properties.* Proceedings TLCA 93, Ultrecht, March 93. LNCS 664, p 328–345.

[14] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions.* PhD thesis, Université Paris 7, January 1989.

[15] E. Poll, P. Severi. *Pure type systems with definitions.* In Proceedings of LFCS'94, Vol 813, pages 316-328, St. Petersburg Russia.

[16] R. Pollack. *The theory of LEGO: a proof checker for the Extended Calculus of Constructions.* PhD thesis, university of Edinburgh, 1994.

[17] R. Pollack. *Implicit Synatx.* Informal Proceedings of First Workshop on Logical Frameworks, Antibes, Mai 1990.

[18] A. Saïbi. *Algèbre constructive en théorie des types, Outils génériques pour la modélisation et la démonstration, Application à la théorie des catégories.* PhD thesis, Paris VI university, forthcoming 1997.

[19] A. Saïbi. *Formalisation constructive de la théorie des catégories.* in preparation, 1996.

[20] A. Tasistro. *Extension of Martin-Löf theory of types with record types and subtyping.* Draft paper, 1993.