Ambiguous Typing

Loïc Pottier

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Ambiguous Typing

Loïc Pottier

N° 6041

Décembre 2006

Thème SYM

ISSN 0249-6399

Ambiguous Typing

Loïc Pottier*

Thème SYM — Systèmes symboliques
Projets Marelle

Rapport de recherche n° 6041 — Décembre 2006 — 11 pages

Abstract: We propose a method to approximate the language of mathematicians by using amgiguities in the notations, in the context of type theory

Key-words: type theory, formalization, ambiguities, notations

* projet Marelle, INRIA Sophia Antipolis

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Typage ambigu

Résumé : On propose une méthode pour approcher le langage math'ematique usuel à l'aide d'ambiguïtés dans les notations, dans le cadre de la théorie des types

Mots-clés : théorie des types, formalisation, ambiguïtés, notations

1 Introduction

There is still a large gap between proving languages used in proof assistants and the usual mathematical language. It is not surprising, because proving languages are designed to be understood by a machine, while mathematical language is designed to be understood by mathematicians ans more generally scientists. But if we want to enlarge the field of application of proof assistants to a public of mathematicians, we have to work to fill this gap. It is what we begin to do in this report, which describes a way to introduce and solve ambiguities in type theory. The two ingredients are overloading and coercions. Overloading allows to give several meanings to the same word. For example, + can denote the addition of integers and the addition of complex functions. Coercions are symbols that are not written, but implicites. For example the multiplication in $b^2 - 4ac$ is a coercion. Or the injection i of R in $R \to R$ in the expression $f + 0$, where f is a real function: we understand here $f + i(0)$.

We will use two languages. The first is a restriction of the language of the proof assistant Coq (see [Coq2006] and [Bertot and Castéran 2004]), which is a type theoretic formal language. We note it also `Coq`. The second is called `CW`, and is also type theoretic. Both are composed by words, applications, abstractions, and dependent products/types. Examples of expressions of these two languages: `Prop`, `fun x:R => (sin x)` , `forall x:N, x=x`.

`CW` will be used to write ambiguous expressions, as close as possible to mathematical expressions (at least those that can be written with an ascii alphabet). And `Coq` will be used to give precise meanings to expressions of `CW`.

2 Overloading

Overloading is defined by an application `ov` from words of `Coq` to words of `CW`. It is extended to the whole language by the rules:

```
   ov(f(a)) = ov(f)(ov(a))
ov(fun x:E => t) = fun ov(x):ov(E) => ov(t)
ov(forall x:E, t) = forall ov(x):ov(E), ov(t)
```

Of course, `ov` is not injective: the problem will be, given a term `t` of `CW`, to determine what are the terms `u` of `Coq` such that `ov(u)=t` and `u` is typable. If several such terms exist, a second problem occurs, which is to choose between them.

3 Coercions

Coercions are functions that are not visible. As multiplication in mathematics, or canonical injection of subsets. We distinguish two kinds of coercions:

argument coercion : if `i` is an "argument coercion", the term `f(i(a))` will be noted `f(a)`.

functionnal coercion : if `c` is a "functionnal coercion", the term `c(x,y)` will be noted `x(y)` or `xy`.

We define coercions as words `c` of `Coq` such that `ov(c) = coercion` (for argument coercions), or `ov(c)= AP` (for functionnal coercions).

Then we define an application `co` from `CW` to `CW` which erase coercions:

`co(coercion(a)) = co(a)`

`co(AP(x,y)) = co(x)(co(y))`

in other cases:

`co(f(a)) = co(f)(co(a))`

```
co(fun x:E => t) = fun co(x):co(E) => co(t)
co(forall x:E, t) = forall co(x):co(E), co(t)
coercion(a) = a
```

4 From `Coq` to `CW`: ambiguous typing

To use both overloading and coercions, we define the application `cw = co o ov`, from `Coq` to `CW`. This application `cw` will be used to print `Coq` terms in a way that they will be more usual to a mathematician. For example the term `forall f:R->R, (plusfun f (constanttofun (ZtoR (NtoZ zeroN))))=f` will be send by `cw` to the term `forall f:R->R, f+0 = f`

The converse is much more difficult: given a term `t` of `CW`, find a(the) term(s) `u` of `Coq` such `cw(u)=t` and `u` is typable. If such a term `u` exists, we will say that `t` is "typable" in `CW`.

The problem is to translate an expression given in a "classical" mathematical language into a valid term for a proof assistant. In fact, this translation is not necessarily unique, and it can even exists an infinity of translations, i.e. typable terms `u` such that `cw(u)=t`. We should be able to enumerate all these terms, trying to give the more interesting in first.

In fact, the problem comes from coercions. They are useful but can lead to serious difficulties. For example, every constant can be viewed as a constant function. This is done by defining a functionnal coercion `c` whith the type `forall E F:Type, F-> E -> F`. But then, if `a` and `b` are arbitrary typable in `CW`, then `a(b)` is also typable! Because `cw(c(a,b)) = a(b)`.

More, if the function `c` is also an argument coercion, then `a(b)` is also the image by `cw` of the terms `c(a,c(b))`, `c(a,c(c(b)))`, etc.

But, for a mathematician, this coercion is natural. So we have to adapt to this situation.

4.1 Typing procedure

Given a term `t` in `CW` and a context Γ (i.e. a list of variables with their types), we will compute `types(t,`Γ`)`, the list of all typable terms in `Coq` whose image by `cw` is `t`, following this process:

- if `t` is a constant, we return the list of constants `u` of `Coq` such that `cw(u)=t`. In practice this list is finite. We order this list by age of the constants: the newest is in first position.
- if `t` is a variable of Γ, we return `t` with its type in Γ.

- if `t` is an abstraction: `t=fun x:E-> a`, for each `F` such that `cw(F)=E` and `F` is a type, we enrich the context Γ with `x:F`, then compute `types(a,`$\Gamma \cup$ `{x:F})`. Finally we return the list of all pairs `(fun x:E-> u) :( forall x:F, T)`, where `cw(F)=E`, `cw(u)=a` and Γ`,x:F`$\vdash$ `u:T`.
- if `t` is a product: `t=forall x:E, a`, we follow the same procedure as abstractions, except that we retain only `Coq` terms whose type is a *sort* (Prop, Set or Type). Same procedure also if `t` is a non-dependant product: `t=E-> F`.
- if `t` is an application: `t = f(a)`, then we return the list of all pairs `g(u):T` where `cw(g)=f`, `cw(u)=a` and $\Gamma \vdash$ `g(u):T`. If this list is empty we change `t` by `f(coercion(a))`. If the result is still empty, we try `f(coercion(coercion(a)))`, etc, until a depth of `n` coercions, with `n` fixed, to insure the termination of the process. If still the result is empty, then we try with the term `AP(f,a)`

Several complex problems come from this procedure:

4.2 Nested coercions

First, we need to bound the depth of nested coercions:
`f(coercion(a))`, `f(coercion(coercion(a))`, etc,
because if not, the list of results would be infinite, and the procedure would not terminate. In practice, this bound can be set to 6, which seems to be sufficient.

Another solution, more complete, to avoid to deal with infinite lists of results, would be to enumerate in breath-first-search, or with a mixed method. For example, to catenate two infinite lists, it would be possible to look at several elements of the first list, then look at the second, and come back to the first, in a complete strategy... We have not experimented this solution.

4.3 Large results: lazy lists

The practice shows that the list `types(t,`Γ`)` can be very large. And the best term is almost always the first[1]. So a solution to have an acceptable computation time is to use *lazy* lists. In the language Ocaml, this is done with this type:

```
type 'a pplist =
 |Vide
 |Retard of (unit -> ('a plist))
 |Cons of ('a * ('a plist))

and 'a plist = ('a pplist) ref
```

[1] the reason is mainly because we look first to newests constants in the resolution of overloading

The use of references allows to modify a list in place as soon as an element is computed. Then a second access to this element will need no computation. Note that the use of lazy lists allows to remove the bound on nested coercions, and then we can work with infinite lists, provided that we work with a complete strategy to deal with concatenation of infinite lists.

5 An example in WCW

To illustrate the use of ambiguous typing, we will take the example of groups and subgroups. This example is done in WCW[2], a wiki which uses a server implementing the ambiguous typing algorithm.

5.1 WCW: a wiki to do mathematics

First let us describe briefly the wiki WCW. A wiki is a web site where users can write and modify easily web pages. More, in WCW, one can include *actions*, which ask a server to perform computations, by calling a web server, for example printing $LaTeX$ expressions, and typing.

To define a new constant of `Coq`, we use the action `def`, with the following syntax:

```
((def nom="+"
      nomcoq="plusR"
      type="R->R->R"
      equations="qs  x y:R, x+y=y+x ..."))
```

This means that `plusR` is a new constant of `Coq` with type `R->R->R`, such that `cw(plusR)= +`.

Note that WCW is developped in french (for the moment), and `forall` is then noted `qs` (for "quelque soit") in `CW`.

All terms in this definition are in `CW`, except the term in the field `nomcoq`, which is in `Coq`.

We have to verify that all these terms are typable:

- `R->R->R`: suppose that we have `cw(R)=R`, then `cw(R->R->R)=R->R->R`. The term `R->R->R` is typable in `Coq`, so it is also typable in `CW`. Note that if a constant of `CW` has no explicit preimage in `Coq`, we suppose that it is a fixpoint of `cw`, as `R` is.
- `qs x y:R, x+y=y+x`: here, knowing that `cw(=)= =`, following the ambiguous typing algorithm we find that `cw(forall x y:R, (plusR x y) = (plusR y x))= (qs x y:R,x+y=y+x)`.

So, in the Coq system, this action would be translated in:

[2] `http://pcmath165.unice.fr/wcw/spikini` Developped in a collaboration with the Laboratoire J.A.Dieudonné, University of Nice (A.Hirschowitz and JP. Giacometti)

```
Parameter plusR:R->R->R.
Axiom eq_plusR1: forall x y:R, (plusR x y) = (plusR y x).
...
```

Other actions exist, to define axioms, to prove theorems interactively, to do exercices, to load libraies, etc.

Now we study the example.

5.2 Groups

First we define the set of groups:

```
((def nom="groupe" nomcoq="groupe_type" type="Ens"))
```

This definition says that `cw(groupe_type)=groupe`. Note that `Ens` is a constant of `Coq` which is no more than `Type`, the type of types. Sets are then indentified to types.

To express the fact that a group is a set, we need a coercion from `groupe` to `Ens`:

```
((def nom="coercion" nomcoq="groupe_ens" type="groupe->Ens"))
```

This means that `cw(groupe_ens)=coercion`.

Now, we define the law of a group:

```
((def nom="*"
      nomcoq="loigroupe"
      type="qs G:groupe, G->G->G"
      equations="qs G:groupe, qs x y z:G, x*(y*z) = (x*y)*z"))
```

With this definition, things are not so trivial. First we define `cw(loigroupe)=*`. Then we have to type `qs G:groupe, G->G->G`, which needs to type `groupe`.

We have `cw(groupe_type)=groupe`, and `groupe_type:Ens` in `Coq`. So `groupe_type` is then a type, and the quantification is correct.

Then we have to type `G->(G->G)` in the context `G:groupe`. Now, there is a problem, because `G` is not a type, since `groupe_type` is not a type of type. So we will try to type `coercion(G)`, to obtain a term whose type is a type of type (the ambiguous typing algorithm is easily adapted to this case of contrained typing). This is possible with the term `groupe_ens(G)` whose type in `Coq` is `Ens`.

Then we can conclude that

`cw(forall G:groupe, (groupe_ens G)-> (groupe_ens G)->(groupe_ens G))`
`= (qs G:groupe, G->G->G)`

For the typing of `qs G:groupe, qs x y z:G, x*(y*z) = (x*y)*z`, we obtain the term `forall G:groupe_type, forall x y z:(groupe_ens G), (loigroupe x (loigroupe y z)) = (loigroupe (loigroupe x y) z)` .

We define now identity element and inverse:

```
((def nom="e"
      nomcoq="elementneutregroupe"
      type="qs G:groupe, G"
      equations="qs G:groupe, qs x:G, x*e=x  ... "))

((def nom="inv"
      nomcoq="inversegroupe"
      type="qs G:groupe, G->G"
      equations="qs G:groupe, qs x:G, x*(inv x) = e ..."))
```

Note that in general, quantified arguments are dropped in `CW` terms, because they can almost always be inferred from the context. But in the case of constants, as the identity element, there are added in the `Coq` term, to allow Coq to succeed in typing the term. A more general solution would be to use in Coq all arguments, even implicit ones, but this needs to redo in ambiguous typing the synthesis of implicit arguments, which can already be done by the Coq typer.

Now, to define subgroups, we define the set of subsets of a set:

5.3 Power set

The set of all subsets of a set:

```
((def nom="partie" type="Ens->Ens" nomcoq="partie"))
```

A subset of a set is also a set. This is a coercion:

```
((def nom="coercion"
      type="qs E: Ens, (partie E) ->Ens"
      nomcoq="partie_to_ens"))
```

We use also a coercion to express the canonical injection of a subset into the whole set (note that this coercion is not possible in the Coq system):

```
((def nom="coercion"
      type="qs E: Ens, qs P: partie E, P-> E"
      nomcoq="partie_to_tout"))
```

The predicate $x \in P$:

```
((def nom="_appartient_a_"
      type="qs E: Ens, E-> partie(E)  -> Prop"
      nomcoq="app"   ))
```

A subset is also a predicate:

```
((def nom="AP"
      type="qs E:Ens,  partie(E) -> E -> Prop"
      nomcoq="abrite"
      equations="qs E: Ens, qs P: (partie E),
                 qs x:E, P(x) = (x _appartient_a_ P) " ))
```

This last constant is a functionnal coercion. It allows to type the term `P(x)` in the equation `qs E: Ens, qs P: (partie E), qs x:E, P(x) = (x _appartient_a_ P)` by typing in fact the term `AP(P,x)`, with the term `(abrite P x)` of `Coq`.

The full subset, defined as a coercion:

```
((def nom="coercion"
      type="qs E:Ens, partie(E)"
      nomcoq="tout"
      equations="qs E: Ens, qs x:E, x _appartient_a_ E" ))
```

Now we can define subgroups.

5.4 Subgroups

The set of subgroups of a group:

```
((def nom="sous_groupe"
      nomcoq="sousgroupe"
      type="groupe->Ens" ))
```

A subgroup is a subset of the group:

```
((def nom="coercion"
      nomcoq="sousgroupe_partie"
      type="qs G:groupe, (sous_groupe G) -> (partie G)"))
```

Definition of a subgroup:

```
((def nom="_est_un_sous_groupe"
      nomcoq="estunsousgroupe"
      type="qs G:groupe, (partie G)->Prop"
      equations="qs G:groupe, qs H:(partie G), (_est_un_sous_groupe H) =
                 ( (e app H) et (qs x y:G, (x app H)
                 et (y app H) -> (x*(inv y)) app H))"))
```

With these definitions we can type this term:
`qs G :groupe, qs H :sous_groupe(G), qs x :H, x*e=e`
with the `Coq` term :

```
 forall  G:groupe, forall  H:(sousgroupe G),
 forall x:(partie_to_ens (sousgroupe_partie H)),
  (loigroupe (partie_to_tout x)  (elementneutregroupe _))
  = ( elementneutregroupe _)
```

Adding several hypothesis, we shall be able to prove:
`qs G:groupe, qs H:sous_groupe(G), (_est_un_sous_groupe H)`,
which become in `Coq`:

```
forall G:groupe, forall H:(sousgroupe G),
(estunsousgroupe (sousgroupe_partie H))
```

5.5 Other examples

The previous detailed example shows that ambiguous typing allows to use classical mathematical notations to deal with sets, subsets, algebra, etc, and in fact to use the same notation for several different objects. This is the case with constants, thanks to overloading, but it is also the case with variables and complex terms, thanks to coercions. More, implicit notations, which are usualy deleted in classical mathematical notations, are possible thanks to functionnal coercions (which can be understood also as overloading of the application). Other examples are available on the wiki WCW[3], for example, vector spaces, mesure theory, lambda-calculus, etc.

6 Conclusion

There are two parts in a work of formalization of mathematics: definitions and proofs. We think that using ambiguities in the context of type theory can be useful to approach the classical mathematical language, and then make more easy the definition step. The example of subgroups shows this fact. It contains the example of subsets, which is a classical problem in formalizations.

We will now investigate the case of ambiguities in proofs. In this case, we will be faced to unification modulo equations, or at least application of proof tactics modulo equations. Indeed, when we use coercions, we often has to say that some compositions are equal. For example, we have `ZtoR o NtoZ = NtoR`, when these three maps are injections in sets of numbers. And we should be able to unify, or at least filter these two terms: `NtoR(n)` and `ZtoR (NtoZ(1))`.

References

[Bertot and Castéran 2004] Yves Bertot, Pierre Castéran "Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions", Springer Verlag, EATCS Texts in Theoretical Computer Science, ISBN 3-540-20854-2

[3] `http://pcmath165.unice.fr/wcw/spikini`

[Coq2006] The Coq proof assistant `http://coq.inria.fr/coq-eng.html`

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399