Ideas over terms generalization in Coq

Vincent Siles[1,2]

LIX/INRIA/Ecole Polytechnique
Palaiseau, France

Abstract

Coq is a tool that allows writing formal proofs and check their correctness in its underlying logic framework, the *Calculus of Inductive Constructions*. Coq's type system handles dependent types, so we should be able to write dependently typed programs with it. However writing complex programs with their full specification can be a rather difficult exercise. In this position paper, I discuss to what extent we can use the mechanism already implemented in Coq to make the work of the programmer easier, especially with the use of Coq's unification algorithm.

Keywords: type inference, generalization, dependently typed programming

1 Introduction

Coq's underlying logic framework, the *Calculus of Inductive Constructions* [1] is a strong basis for using dependent types. It is a powerful way to write regular programs along with a specification. However, this exercise can become arduous if you want a precise specification or if you are using complex proofs. In this position paper, I would like to show to what extend we can use the mechanism already implemented in Coq to ease the work of the programmer, in particular with the use of Coq's unification algorithm to compute missing terms.

The main idea is to be able to write ML-like program in the more expressive type system of Coq, with the same support for type inference and polymorphism that we can find in *OCaml* or *MLF* [2], where the computer can, to some extent, infer some type information omitted by the programmer.

2 Using Coq's unification to compute omitted terms

Implicit syntax and unification

In CIC, every binder is *explicit*. All of the implicit syntax mechanism of Coq is just syntactic sugar to be more user-friendly, which mainly add holes (also called *ex-*

[1] Thanks to my supervisors Hugo Herbelin and Bruno Barras

[2] Email: vincent.siles@lix.polytechnique.fr

istential variables or *evars*) in the input term, expecting that the unification will fill them. In other languages like the variant of Calculus of Implicit Constructions [4] with decidable type-checking [5], the implicit syntax mechanism is in the core language, but the user still has to point out which argument of his function can be left implicit.

My claim is that some of those implicit arguments can be automatically guessed by the computer, and so the user can omit them entirely from his code. This however, requires higher-order unification which is a complex problem with the rather unfortunate property of being undecidable. Hence we tried, using an experimental test version of Coq (based on the version 8.1), to see to which extent Coq's algorithm can "guess" this information.

Type inference

As mentioned above, all the binders are explicit. The consequence is that type inference will not add binders to the terms entered by the user, it will only try to compute missing information from implicit syntax hints. We have implemented the ideas of this paper in a test version of Coq to mimic the behaviour of ML-languages. We will now have a close look to some basic example, to understand how Coq deals with this problem.

```
Coq < Definition id x :=x .
Error: Cannot infer a type for x
```

If we check the internal representation of the term above, we will see that Coq added an evar for the type of `x` and failed to fill it. But it is clear here that any type could have been used to fill this hole, because this hole is not defined yet, and has no constraint on it (except being a valid type, a constraint we will omit to mention henceforth). As a first try, we could just add enough binders in front of a term with holes to bind those *constraint-free holes* and obtain the following behaviour:

```
Coq < Definition id x :=x .
id is defined
Coq < Print id.
id = fun (T : Type) (x : T) => x
     : forall T : Type, T -> T
```

Now we will consider a new example : the application function.

```
Coq < Definition app f x := f x.
Error: Cannot infer a term for an internal placeholder
```

This time, the problem is more subtle and is a consequence of the property of subtyping in Coq's type system. The system will manage to add some evars and will have (approximately) the following constraints [3] :

$$[\,] \vdash f : ?a \to ?b \quad [f : ?a \to ?b] \vdash x : ?c \quad [\,] \vdash ?c \leq ?a$$

We only require x's type to be a subtype of f's domain. So the previous approach is broken because it would build the term

[3] $?c \leq ?a$ stands for "$?c$ is a subtype of $?a$"

`Definition app A B (f:A->B) x := f x` which has still the constraint over `x`'s type to be solved. This is were the subtyping system comes handy. Thanks to the subtyping rules[4], we can prove that if you are a subtype of a variable, then you are equal to this variable. So, we can generalize $?a$ and $?b$ into type variables A and B, so that the remaining constraint is changed to $[] \vdash ?c = A$ and can easily be solved. Our `app` function can now be computed with its full type information:

```
Coq < Definition app f x := f x.
app is defined
Coq < Print app.
app =
fun (T T0 : Type) (f : T -> T0) (x : T) => f x
     : forall T T0 : Type, (T -> T0) -> T -> T0
```

From this example, we learned that it is necessary to collapse all the constraint-free variables that are related by subtyping relations. This can be easily done if we look at the evars to be the nodes of a graph where the edges are the subtyping constraints: every connected graph stands for one variable we can bind in front of the term.

Recursive functions

With non-dependent types, the case of recursive functions works fine. Since we have no dependencies to deal with, we can add the binders *before* the fixpoint definition to close the term without interfering with the recursive definition. This will not be the case with dependent types, as we will see later.

```
Coq < Fixpoint length l :=
Coq < match l with
Coq < | nil => 0
Coq < | cons _ l' => 1+ (length l')   end.
Error: Cannot infer a term for an internal placeholder

 (* The test version should have produced the following term *)
Coq < Definition length A  := fix length (l:list A) :=
Coq <  match l with
Coq <  | nil => 0
Coq <  | cons _ l' => 1 + (length l')   end.
length is defined
```

Generalizing into ML

Most ML type inference algorithm have a *gen* rule (e.g. the Hindley-Milner Algorithm, or MLF's *gen* rule [2]), which is related to the use of implicit products in the langage. But Coq does not have such rule or product. Here we saw that we can postpone this step to be the very last and mimic the *gen* rule just by adding binders for every constraint-free evars (with respect to the subtype relation). Thanks to this, we can now type directly in Coq our favorite *OCaml* programs, with very little syntactic changes, even for recursive functions.

[4] http://coq.inria.fr/V8.1pl3/refman/Reference-Manual006.html#toc26

3 Coq: a dependently typed programming language ?

The next step is to extend the previous algorithm to dependent types. We will consider two examples of programs that we would like to be type-checkable in a version of Coq with implicit generalization .

```
Coq < Print vector.
Inductive vector (A : Type) : nat -> Type :=
    Vnil : vector A 0
  | Vcons : A -> forall n : nat, vector A n -> vector A (S n)
```

Length

```
 Fixpoint length v := match v with
  | Vnil => 0
  | Vcons _ _ w => 1 + (length w)   end.
```

The first function is a dumb length function which computes the length of a vector as if it were a list: we do not care about the length information in the type. With the approach of the previous section, we can successfully fill the holes and type check the term, but a major change occurs: adding binders in front of the term is not enough, now we have to modify the structure of the term itself to take into account the dependency of `v` over its length.

```
Coq < Definition length A n := fix length (v:vector A n) {struct v} :=
Coq <   match v with
Coq <    | Vnil => 0
Coq <    | Vcons _ _ w =>1+(length w)   end.
Toplevel input, characters 127-128
>    | Vcons _ _ w =>1+(length w)
>                               ^
Error:
In environment
n : nat
A : Type
length : forall (v : vector A n) , ?3
v : vector A n
a : A
n0 : nat
w : vector A n0
The term "w" has type "vector A n0" while it is expected to have type
 "vector A n"
```

This first definition is the case of a global generalization, and will be ill-formed because of the recursive call, since the length of `w` is not `n` but `n-1`.

```
Coq < Definition length A := fix length n (v:vector A n) {struct v} :=
Coq <   match v with
Coq <    | Vnil => 0
Coq <    | Vcons _ _ w =>1+(length _ w)   end.
length is defined
```

The second one is a local generalization, which will preserve the link between `v` and its size.
So we have to be careful were we put the binders, and we have to also change the recursive call (you can notice that we add an _ in the recursive call) to take into account the new argument. This is the most dangerous step because if we change the structure of a term, we may break several constraints or results already computed by Coq, so we have to redo the inference step from the beginning, which may be quite costly.

With this enhancement of the algorithm, however, the length function can be defined as we desired. Note also that the length, which is not used in the computation, has been inferred.

Append

```
Fixpoint append v w {struct v} := match v with
  | Vnil => w
  | Vcons a _ v' => Vcons a _ (append v' w)   end.
```

The second function takes two vectors and puts the second at the end of the first one to build a longer vector. Again this time, the length of the vectors are not used in the computation, only to type check the recursive call. This time, however, the right length is not available in the context of the evar as it was in the *length* function; it has to be computed from the length of `v'`, `w` and the `plus` function. This is the kind of problem that will break our algorithm, because Coq v8.1's unification algorithm only knows how to solve equations of the form $?x\ x_1 \ldots x_n = x_i$, while here one needs to solve equations of the form $?x\ O = a$ and $?x\ (S\ n) = b$: it will not be able to reconstruct the *plus* function for natural numbers from scratch.

There is also a hidden problem in the typing of the match, in its return type. If we do not help Coq here, it will infer the return type to be the type of w instead of being of type *vector* p where p is the sum of the length of v and w (this problem has more to do with typing dependent match and is out of our scope here), hence failing to infer the right return type, leading to an ill-typed term.

In this case, the only solution we could think of has been to provide the return type of the function and the return argument of the match by hand, so that Coq will manage to infer the right return type and also the use of the `plus` function in the recursive call.

```
Coq < Fixpoint append (A:Type) n (v:vector A _)
                                 m (w:vector A _) : vector A (n+m) :=
Coq < match v  in (vector _ p) return (vector A (p+m)) with
Coq < | Vnil => w
Coq < | Vcons a n' v' => Vcons _ a _ (append _ n' v' _ w)   end.
append is recursively defined
```

But in so doing we have given almost all of the typing information of the term, precisely what we wanted to avoid in the first place.

4 Conclusion and Future work

The non-dependent part of this attempt at an algorithm has been partially implemented in an experimental extension of Coq (everything but the recursive function support, due to the complexity of Coq's fixpoint handling) and is working well, but since the dependent part is really not the good way to solve this problem, this implementation has been dropped.

The two main problems of type inference we spotted are quite different, but they are both important open problems concerning programming with dependent types:

- Inferring terms built from the ones in the context: we could do simple arithmetic but something more robust like the *Program tactic* [6] may be a better one.
- Dependent matching: inferring the return clause of a match, or getting all the information we can from the pattern matching is currently under discussion among the Coq Development Team. *Epigram* [7] has a rather interesting solution but it still requires some work from the user. We still hope to infer some of those hints automatically.

References

[1] Coq Development Team, *The Coq Proof Assistant Reference Manual*, URL:http://coq.inria.fr/V8.1pl3/refman/index.html.

[2] D. Rémy, D. LeBotlan, B. Yakobowski, *MLF*, URL:http://pauillac.inria.fr/ remy/mlf/.

[3] Ulf Norell, *Towards a practical programming language based on dependent type theory*, Phd thesis, Chalmers University of Technology, 2007.

[4] A. Miquel, *Le Calcul des Constructions implicite: syntaxe et smantique*, Phd thesis, Université Paris 7, 2001.

[5] B. Barras,B. Bernardo, *The Implicit Calculus of Constructions as a Programming Language with Dependent Types*, FoSSaCS (2008), 365–379.

[6] Matthieu Sozeau, *Russell and the tactic Program*, URL:http://mattam.org/research/russell.en.html.

[7] H. Goguen, C. McBride, J. McKinna, *Eliminating Dependent Pattern Matching*, Essays Dedicated to Joseph A. Goguen (2006), 521–540.