Proof assistant project

Elsa Lubek

December 2024

1 Introduction

Here is a description of the proof assistant that I implemented using Ocaml as part of a computational logic course. I followed the questions of a lab and completed all the mandatory parts (except for the proof of commutativity and associativity of multiplication). All my tests work, but most of them are written as comments (you can uncomment them to try them)." There were two different parts: in the first, I implemented a proof assistant using Simple type. In the second part, I programmed one using Dependent type. I decided to explain the difficulties related to the theory. In the third part of this report, I will highlight some general programming difficulties that I encountered. In the fourth, some arbitrary choices.

2 Simple

First, I had to implement a proof assistant using simple types. I had to define a type for expressions and a type for types. The idea of Curry-Howard correspondence is that a formula is true as long as it corresponds to the type of a well-typed term (under a given context). So I had to define what expressions are, and how to infer their type (with the infer function). My infer function and check function are mutually recursive. I used Church style.

There is more freedom in the choice of implementation for expressions than what we could think of. Indeed, many expressions are not well-typed (even though only those which will be are eventually interesting). So we can put some constraints during the formation of the expressions or alternatively allow more expressions but be more careful when inferring the potential type. I faced such a choice of implementation when implementing the coproduct. Indeed, I first used the syntax :

```
Case of tm*tm*tm
```

where I knew that the second and third terms had to be lambda-terms for the formula to be well-typed, but it was the job of my infer function to check it. However, another possibility, which I finally chose because the parser provided in the lab was more compatible with it was to encode it as such :

```
Case of tm*var*tm*var*tm
```

In the theory, it made no difference because lambda terms are of the form $var * tm$.

At first, I did not really understand how to implement false intelligently. I created a type for false (called Zero) but then, when it came to inferring type, I thought that the fact that false implies anything meant that checking whether something of type false is of type A should return true (for any A). It was not philosophically what we would expect, but in practice, it worked quite efficiently. For real, false implies A means that I have an explicit term of type $\bot \to A$ which is ($\bot$-elim).
So when I defined the lambda term $\lambda x^{Zero}.x$, at any time when I was checking that it was of type $\bot \to A$ it was true, so I could use it like ($\bot$-elim). In other words, I could recover case(A) by $\lambda x^{Zero}.A$ and did not add any formula that would not correspond to a term in the right implementation, which is why I did not see my error immediately (although I was unsure of what I was doing).
Indeed, because there is no term of type Zero, any occurrence of Zero comes from a lambda term. Let's take a formula well typed in my previous implementation. If I use the fact that x is of type A in the type checking, I can replace x by a ($\bot$-elim A x) and build a term of the similar type.
Anyways, When I understood my mistake, I correctly implemented the "Case ty". Thinking about it made me understand better the idea of the implementation of simple types. In particular, types are not supposed to be included in others in the sense of check. Although I could prove the same formulas (because I was only interested in the infer function), my check function did not behave as expected.

3 Dependent

In this part, I implemented a proof assistant using dependent types. The new elements were induction and equality. My new assistant was more powerful in the sense that I could do induction and prove equalities (so for instance arithmetic properties). However, I did not reimplement coproducts, for instance, nor did I add tactics to assist during the proof because I preferred to focus on new theoretical aspects. I could point out that my prover is inconsistent because Type is of type Type. However, it is still interesting because, with more time, I could have implemented universes and, with minor modifications, built a consistent model.

In this part, I had to handle the fact that infer should call normalization. My check function was checking that a term was of a given type but I also needed to use at some points the function "conv" itself to compare terms (up to α-conversion).

I had an issue with infinite loopings when implementing induction. Initially, I had :

```
| Ind (p, z, s, m) -> (check ctx z (App(p, Z));
  match infer ctx s with
    | Pi (n, Nat, f) ->
        let pn = normalize ((n , (Nat, None))::ctx) (App(p,Var n)) in
        let pSn = normalize ((n , (Nat, None))::ctx) (App(p, S n)) in
         if (conv ((n , (Nat, None))::ctx) (f (Pi("x", pn, pSn)) then
           (
        (check ctx m Nat; normalize ctx (App(p,m)))
         )
         else raise(Type_error(to_string(s)^"is not a function"));
    | _ -> raise (Type_error("Your recursor is not well typed"))
```

However, the variable n could have another meaning in the definition of p, and list.assoc was not selecting the correct n when inferring the type of App(p, Var n), so I had to replace it with a fresh variable.

```
| Ind (p, z, s, m) -> (check ctx z (App(p, Z));
  match infer ctx s with
    | Pi (n, Nat, f) ->
        let fresh_n = fresh_var () in
        let pn = normalize ((fresh_n , (Nat, None))::ctx) (App(p,Var fresh_n)) in
        let pSn = normalize ((fresh_n , (Nat, None))::ctx) (App(p, S (Var fresh_n))) in
        if (conv ((fresh_n , (Nat, None))::ctx) (subst n (Var fresh_n) f) (Pi("x", pn, pSn))) then
          (
        (check ctx m Nat; normalize ctx (App(p,m))) (*just for beauty*)
         )
         else raise(Type_error(to_string(s)^"is not a function"));
    | _ -> raise (Type_error("Your recursor is not well typed"))
```

In the dependent case, I also had to handle values. A term such as Var x in context (x, (Var x)) would make my program loop so I added a specific case for that.

4 Difficulties

I would like to summarize some difficulties linked to the simple fact of implementing in general, not specifically linked to logic theory.

First, when I was coding, I sometimes thought I could try something if I was not sure and see later during tests if it led to bugs. This is completely contradictory with the philosophy of logic and proof assistant. I realized that some mistakes can lead to bugs way later and quickly decided to change my approach. For instance, I forget to add "()" in my definition of fresh (the function creating a fresh variable) and it was always returning x1. Curiously, it was not leading

to any bug before some quite sophisticated proofs. It is better to take time to be sure about what I am doing.

Second, when trying to debug my code, it was sometimes hard to find out where the problem came from in my code. I had to add very precise error messages. In particular, in the part 5 (dependent) when I was using the prover, I did not know whether the mistake came from the proof I was giving or the prover itself. I had to create terms of intermediate difficulty. Both case actually happened. For instance, I once spelled "succom" instead of "succomm" and my program was looping forever. Another time, the program was looping because of my infer function.

5 Implementation choices and possible extensions

Here are some little initiatives I took : -In the simple part, I decided to name differently my witnesses in the two cases ("fA" and "fB") but it was not necessary.

```
|Coprod(tyA,tyB) -> (let t1 = prove (("fA",tyA)::env) a
and t2 = prove (("fB",tyB)::env) a in
     Case(f , "fA", t1 , "fB", t2) )
```

-In the dependent type part, I created a function that prints only the beginning of the context because it contains the most recent variables (otherwise it quickly becomes very long).
-I really have plenty of ideas of extensions :

- Apart from the one that are suggested in the lab, I think it could be very useful to add a command "comment" to create sections within a proof.
- I could create a more subtle substitution using a function that look for free variables. I could also change my function freshvar to reuse variables that have not been used (variables that do not appear in the context).
- I could also provide more help by suggesting tactics....

...but all of that already exist, it is just Agda !

6 Conclusion

To conclude, it was particularly interesting to program from (almost) scratch a proof assistant. First, since I am now used to using one (Agda), I can better understand its behaviour. Secondly, it also helped me better understand the course.