Generalization in type theory based proof assistants

Olivier Pons*

Departamento de Informática, Universidade do Minho, Portugal
pons@simple-card.com

Abstract. This paper describes a mechanism to generalize mathematical results in type theory based proof assistants. The proposed mechanism starts from a proved theorem or a proved set of theorems (a theory) and makes it possible to get less specific results that can be instantiated and reused in other contexts.

1 Introduction

In mathematics, it is common to try to generalize previously obtained results in order to reuse them in other contexts and to get more structured representations. To do so, we have to find which parts of a proof can be generalized, we have to understand the analogies between different mathematical structures and the analogies between proofs on these structures, and finally to study how these similarities can be useful in the development of new theories. For example, the basic results of arithmetic can be generalized to get algebraic theories like group or ring.

Today, mechanical theorem provers are increasingly used for large-scale formalizations of mathematics and programming languages semantics or for algorithms certification and it seems interesting to investigate how the generalization process used in mathematical development can be used in this context.

Generalization and reasoning by analogy have been intensively studied in Artificial Intelligence and in automatic proof systems (in which they are used to generate intermediate lemmas in order to avoid diverging proof search) but little work has been done in the framework of interactive proof systems based on type theory.

The most difficult task is to detect analogies between objects. The formalization of this problem relies on the notion of anti-unification introduced by Plotkin[Plo69]. The problem is, given two terms t_1 and t_2 to find the tuple (g, θ, σ) where g is a term and θ and σ two substitutions such as $\theta(g) = t_1$ and $\sigma(g) = t_2$. Finding the most specific generalization of tow terms is now a well understood problem for first order theories but not for higher order systems. Contributions to the understanding of the higher order case are for example

* The author is supported by the Portuguese Science Foundation (Fundação para a Ciencia e a Technologia) under the Fellowship PRAXIS-XXI/BPD/22108/99.

Pfenning studies of anti-unification for a subset of the terms of the Calculus of Constructions[Pfe91] or the categorical approach of Hasker and Reddy [HR92] that have shown that in the case of the second order, the complete set of most specific generalizations can be computed.

In this paper we follow Kolbe[KW94] on a more pragmatic way. Our aim is not to find the least general generalization of two terms but simply to help the user to define generalizations of already proved theorems without assuming future instantiations of these theorems.

Our experiments have been done using the Coq system[Ba99] but the idea is generic and could be adapted to any system based on type theory that represent theorems as types and proofs as terms such as Nuprl[Ca86] or Lego[LEG].

Formally our problem can be summed up as follows : We start with a triple (s, t, l) where s is a type, t is a term of type s, and l is a set of free identifiers occurring in s. We want to find the smallest set l' of identifiers containing l, a type s' term t' such that s' and t' come respectively from the abstraction of the identifier of l' in s and t and such that t' be of type s' (The practical meaning of l', s' and t' will be highlighted in the first example).

The first section of this paper is this introduction, the second section show our basic generalization mechanism at works by abstracting a function operator in a theorem. The third section formally describes the mechanism. The fourth section discuss the correctness and the limitations of the basic mechanism and propose a first extension. The fifth and sixth section discuss with examples the abstraction of inductive types and inductive predicates. The seventh section discuss the construction of abstract theories. The eighth examine the abstraction mechanism for inductive objects. The ninth section gives a conclusion and enumerates the points that deserveb a more thorough treatment in further work.

2 A quick overview

2.1 Abstraction

As an example, we will start on a basic lemma about multiplication on natural integers. Suppose that we have proved the following property :

$mult\_permute$:
$\forall\ n, m, p : nat.(\boxed{\texttt{mult}}\ n\ (\boxed{\texttt{mult}}\ m\ p)) = (\boxed{\texttt{mult}}\ m\ (\boxed{\texttt{mult}}\ n\ p))$

We wish to generalize the statement of the theorem $mult\_permute$ to get a new statement in which the operator will not be $mult$ any more but an unspecified binary operator on natural integers. For that we will abstract the identifier $mult$ in the statement $mult\_permute$. The generalized statement[1] we obtain has the form:

[1] in which we denote f the abstracted operator to avoid the confusion with $mult$

```
∀f : nat → nat → nat
   ∀n, m, p : nat <...> ⇒
      (f n (f m p)) = (f m (f n p)).
```

But as highlighted by the square bracket this statement is not true any more in the general case.

The problem is to find the properties that must be verified by the operator f in order to ensure that the new statement is still a theorem. Here, intuitively the statement will remain true for any associative and commutative function on integers but how to find this properties in the general case ? In order to justify our answer let us quickly recall how to work with the Coq proof system. When building a mathematical theory in the Coq proof assistant we start by defining the objects of this theory (They are terms of the Calculation of Inductive Constructions (CCI)). Then we state (in the CCI) some properties of these objects. To prove these properties we have a set of commands (tactics). The application of a tactic to an initial goal generates a list of subgoals. The user then chooses a subgoal in this list and applies a new tactic on this subgoal. The produced subgoals are then inserted in the list of remaining subgoals. The proof finishes when the resulting list is empty. The system then uses the list of tactics to build the proof object (a term of the CCI) and check the complete proof by type-checking this term. So that we have several representations of the same proof. The list of tactics (called a proof script) reflects the strategy of the user and is generally the only thing he is interested in but in the general case the script does not give enough information[2] to support automatics proof analysis and manipulations. Therefore, to find which properties must be checked by the operator which is abstracted, it is necessary to examine the proof term of the initial theorem. We will also use this term to build a proof of the generalized statement. Thus let us observe the proof term of *permute_mult*:

```
λn, n, p : nat
 (eq_ind_r nat (mult (mult m n) p)
    λn : nat (mult m p) = n0
    (eq_ind_r nat (mult n m)
          λn1: nat (mult n (mult m p)) = (mult n1 p)
          (mult_assoc_l n m p)
          (mult m n)
          (mult_sym m n))
    (mult m (mult n p))
    (mult_assoc m n p))
```

In addition to *mult*, *nat*, and =, which already appeared in the initial statement, the free identifiers which appear in the proof are *eq_ind_r*, *mult_assoc_l* and *mult_sym*. Therefore, in order to file out the abstracted statement, it is

[2] mainly because it may contains automatic decision procedures

necessary to determine which of them are related to the identifier *mult* which has been abstracted in the statement.

Thus, we seek their statements (*i.e.* their types) in the current environment. We find respectively:

eq_ind_r : $\forall A : Set.\forall x : A.\forall P : (A \to Prop)(P\ x) \to \forall y : A.y = x \to (P\ y)$
$mult\_assoc\_r$: $\forall n, m, p : nat.(\boxed{\texttt{mult}}\ (\boxed{\texttt{mult}}\ n\ m)\ p) = (\boxed{\texttt{mult}}\ n\ (\boxed{\texttt{mult}}\ m\ p))$
$mult\_sym : \forall n, m : nat.(\boxed{\texttt{mult}}\ n\ m) = (\boxed{\texttt{mult}}\ m\ n)$

It appears that only the last two statements refer to the identifier *mult*. Thus, we deduce, that the only properties of the multiplication which are used in the proof are symmetry and left associativity which correspond to $mult\_sym$ and $mult\_assoc\_l$ [3].

Thus, it is also necessary to abstract these two identifiers in the generalized statement in order to express the constraints on the operator f. We get the following statement:

$$
\begin{array}{l}
Lemma\ \ f\_permute\ \ : \\
\forall f : nat \to nat \to nat. \\
\ \forall f\_assoc : \forall n, m, p : nat.(f\ \ n\ \ (f\ \ m\ \ p)) = (f\ \ (f\ \ n\ \ m)\ \ p). \\
\ \ \forall f\_sym : \forall n, m : nat.(f\ \ n\ \ m) = (f\ \ m\ \ n) \\
\ \ \ \forall n, m, p : nat.(f\ \ n\ \ (f\ \ m\ \ p)) = (f\ \ m\ \ (f\ \ n\ \ p))
\end{array}
$$

The proof of this theorem is obtained by abstracting *mult*, $mult\_sym$ and $mult\_assoc\_l$ in the initial proof term. This leads to the following proof term :

$$
\begin{array}{l}
\lambda f : nat \to nat \to nat. \\
\ \lambda f\_assoc\_l : (n0, m0, p0 : nat)(f\ \ n0\ \ (f\ \ m0\ \ p0)) = (f\ \ (f\ \ n0\ \ m0)\ \ p0). \\
\ \ \lambda f\_sym : (n0, m0 : nat)(f\ \ n0\ \ m0) = (f\ \ m0\ \ n0). \\
\ \ \ \lambda n, m, p : nat. \\
\ \ \ \ (eq\_ind\_r\ \ nat\ \ (f\ \ (f\ \ m\ \ n)\ \ p)\ \ \lambda n0 : nat.(f\ \ n\ \ (f\ \ m\ \ p)) = n0 \\
\ \ \ \ \ \ \ \ (eq\_ind\_r\ \ nat\ \ (f\ \ n\ \ m)\ \ \lambda n0 : nat.(f\ \ n\ \ (f\ \ m\ \ p)) = (f\ \ n0\ \ p) \\
\ \ \ \ \ \ \ \ \ \ \ \ (f\_assoc\_l\ \ n\ \ m\ \ p)\ \ (f\ \ m\ \ n)\ \ (f\_sym\ \ m\ \ n)) \\
\ \ \ \ \ \ \ \ (f\ \ m\ \ (f\ \ n\ \ p)) \\
\ \ \ \ \ \ \ \ (f\_assoc\_l\ \ m\ \ n\ \ p))
\end{array}
$$

2.2 Instantiations:

We now check that we can instantiate $f\_permute$, with addition and the proof of its associativity and the commutativity, to obtain a more specific instance of the theorem. We get:

[3] On our example these two identifiers also appear in the script, however this is not always the case.

```
Lemma plus_permute :
  ∀n, m, p : nat. ((plus n (plus m p)) = (plus m (plus n p))).
Proof (f_permute plus plus_assoc_l plus_sym).
```

In the following section we describe the broad outline of an interactive tool helping the user to carry out such generalizations.

3 A tool to assist the generalization

3.1 Basic principle

Given a statement S and a term P which is a proof of S, the user provides the name of the function that he wishes to abstract in S (to simplify the following explanation we suppose that he selects only one function identifier denoted by f). In a graphical environment such as *CtCoq*[Ba96] or *Proof-General*[Aa99], this can be done by a single mouse click. The following operations should then be made automatically:

1. Find the type t associated with the identifier f.
2. Recover the list of all the free identifiers which appear in the proof term P and do not already appear in the statement S
3. Find the types associated with these identifiers.
4. Among these identifiers, select all those whose type refers to the identifier f that has been abstracted. We will denote f_{P_i} the selected identifiers and t_{P_i} their types (which express the properties of f which have been used).
5. Build the statement of the theorem generalized by abstracting f and the f_{P_i} in S. We obtain a generalized statement of the form:

$$\forall f : t. \forall f_{P_1} : t_{P_1} \ldots . \forall f_{P_i} : t_{P_i} \ldots . S$$

6. Build the proof term associated to this statement by abstracting on P. We get a term of the form :

$$\lambda f : t. \lambda f_{P_1} : t_{P_1} \ldots . \lambda f_{P_i} : t_{P_i} \ldots . P$$

Remark 1. Actually, steps 1 to 4 can be done one time for all by computing the dependency graph of the development. Each node is anotated by an identifier, a type and a term. When abstracting an identifier I corresponding to the node N_i in a statement S corresponding to the node N_s, we should also abstract all the identifiers associated with a node that appears in a path between N_i and N_s.

3.2 Choosing the names of the abstracted identifiers

As we can notice in the statement of the lemma *f_permute* assigned in the previous section, the names given to the abstract identifiers are not randomly

selected. We propose a small algorithm to name the properties of the abstracted terms.

It is usual to express a property of an operator by the name of this operator followed by the name of the property (often separated by a character *underscore*). To express that it corresponds to an unspecified function, we will use f (then f_i if we abstract more than one operator) to denote the abstract operator. Thus, it is also necessary to substitute f to all the occurrences of the abstracted operator in the statement.

Once we have recovered the list of identifiers resulting from step 4 in our algorithm, we use their names to build new names by substituting f to the name of the operator in the old name of the property; thus $mult\_assoc$ becomes f_assoc which remains significant. Then we substitute f to all the occurrences of the abstracted operator in the types of the identifiers recovered at step 3. Finally we substitute f and the names of properties formed from f in the proof term.

4 Correctness problems

Unfortunately all these transformations does not produce a type-checkable term. Actually our generalization mechanism may produce a badly typed term because proof objects are not completely explicit.

In fact for efficiency and clarity reasons most systems add to the traditional deduction rules some computational rules. The application of such a computational rules does not appear in the proof term.

A classical example of such rule is the iota-reduction. Let us consider the definition of the function *plus* using *Fixpoint* and *Cases*

```
      Fixpoint plus λn : nat.  :  nat→ nat  :=
         λm : nat.
         Cases n of
r_1 ☞          O    ⇝ m
r_2 ☞        | (S p) ⇝ (S (plus p m)) end.
```

The *iota*-reduction allow to identify (*plus* 2 2) to 4 without any demonstation, just using the computational rules r_1 and r_2 to reduce (*plus* 2 2).

Now let us show when such reduction step may interfere with our generalisation mechanism. We consider the right associativity of *plus* :

Lemma plus_assoc_r :
$\forall n, m, p : nat.((plus\ (plus\ n\ m)\ p) = (plus\ n\ (plus\ m\ p)))$.

We will see two ways to build the proof. The first one produces a term in which *plus* may be abstracted safely. The second one produces a term for which the abstraction of *plus* brings out a badly typed term. The last subsection proposes a solution for the bad case.

4.1 The correct case

If we prove the statements $plus\_assoc\_l$ by the small script below, using **explicitly** the proof of the left associativity:

```
Intros;Apply  sym_eq; Apply  plus_assoc_l
```

we get the proof term

$$\lambda n, m, p : nat.\\ (sym\_eq\ nat\ (plus\ n\ (plus\ m\ p))\ (plus\ (plus\ n\ m)\ p)\\ \quad (plus\_assoc\_l\ n\ m\ p))$$

We can abstract $plus$ as previously to get a lemma f_assoc_r which proof term is

$$\lambda f : (nat \to nat \to nat).\\ \lambda f\_assoc\_l : ((n, m, p : nat)(f\ n\ (f\ m\ p)) = (f\ (f\ n\ m)\ p)).\\ \lambda n, m, p : nat.\\ (sym\_eq\ nat\ (f\ n\ (f\ m\ p))\ (f\ (f\ n\ m)\ p)\ (f\_assoc\_l\ n\ m\ p))$$

4.2 The problematic case

Now, if we use the script below to build the proof term of $plus\_assoc\_r$

```
Intros;Apply sym_eq;Elim n; Simpl;Auto with *.
```

This will produce a different proof term that does not refers to the left associativity[4]

$$\lambda n, m, p : nat.\\ (sym\_eq\ nat\ (plus\ n\ (plus\ m\ p))\ (plus\ (plus\ n\ m)\ p)\\ \quad (nat\_ind\ \lambda n0 : nat.(plus\ n0\ (plus\ m\ p)) = (plus\ (plus\ n0\ m)\ p)\\ \quad\quad (refl\_equal\ nat\ (plus\ m\ p))\\ \quad\quad \lambda n0 : nat.\lambda H : ((plus\ n0\ (plus\ m\ p)) = (plus\ (plus\ n0\ m)\ p)).\\ \quad\quad\ (f\_equal\ nat\ nat\ S\ (plus\ n0\ (plus\ m\ p))\ (plus\ (plus\ n0\ m)\ p)\ H)\ n))$$

Abstracting $plus$ in this term we can build a term f_assoc_r

$$Definition\ f\_assoc\_r\ :=\\ \lambda n, m, p : nat.\lambda f : nat \to nat \to nat.\\ (sym\_eq\ nat\ (f\ n\ (f\ m\ p))\ (f\ (f\ n\ m)\ p)\\ \quad (nat\_ind\ \lambda n0 : nat.(f\ n0\ (f\ m\ p)) = (f\ (f\ n0\ m)\ p)\\ \quad\quad (refl\_equal\ nat\ (f\ m\ p))\\ \quad\quad \lambda n0 : nat.\lambda H : ((f\ n0\ (f\ m\ p)) = (f\ (f\ n0\ m)\ p)).\\ \quad\quad\ (f\_equal\ nat\ nat\ S\ (f\ n0\ (f\ m\ p))\ (f\ (f\ n0\ m)\ p)\ H)\ n))$$

[4] it does not contains $plus\_assoc\_l$ as free identifier.

But this term is badly typed and a type checker should complain something like :

```
In environment [n : nat; m : nat; p : nat; f : nat→nat→nat]
The term nat_ind of type :
    ∀P : (nat→Prop).(P O)→(∀n : nat.(P n)→(P (S n)))→∀n : nat.(P n)
Cannot be applied to the terms
  λ n0 : nat.(f n0 (f m p)) = (f (f n0 m) p) : nat→Prop
  (refl_equal nat (f m p)) : (f m p) = (f m p)
  λn0 : nat.λ H : ((f n0 (f m p)) = (f (f n0 m) p)).
   (f_equal nat nat S (f n0 (f m p)) (f (f n0 m) p) H)
    : ∀n0 : nat.
        (f n0 (f m p)) = (f (f n0 m) p)
        →(S (f n0 (f m p))) = (S (f (f n0 m) p))
  n : nat
```

In such a case, the transformation is considered as illegal and is just rejected by our basic generalization mechanism.

4.3 How to enlarge the set of legal transformations ?

To get a correct transformation when the proof term produced by the basic abstraction mechanism does not type-check, we will try to replace the badly typed subterm by a variable with the right type that would be abstracted.

For that, we can, during the transformation annotate the subterms with their translated type (ie. the type resulting from the application of the substitution to the inital type). Then each time the type of a translated subterm (the proposed one) differs from the translated type (the expected one), we replace the subterm by a variable with the expected term.

For example the badly typed version of f_assoc_r of the previous subsection becomes :

```
Definition f_assoc_r :=
λn, m, p : nat.
 λf : (nat→nat→nat).
 λp1 : ([n0 : nat](f n0 (f m p)) = (f (f n0 m) p) n).
 (sym_eq nat (f n (f m p)) (f (f n m) p) p1)

: ∀n, m, p : nat.∀f : (nat→nat→nat).
        (λn0 : nat.(f n0 (f m p)) = (f (f n0 m) p) n)
        →(f (f n m) p) = (f n (f m p))
```

This simply expresses that on *nat* left associativity implies right associativity !

Remark 2. Managing such "bad case" not alway produces useful results. The trivial case where the transformation is stupid is when the result is a tautology (of the form $\forall c : tc \ldots \forall c_n : tc_n.tc_n$). We will reject such produced terms.

5 More abstraction

So far, we have limited the abstraction to function identifiers. Let us consider again the example of the section 2.1. The permutation property remains true for any function $f : E \to E$ associative and commutative, whatever the set E. Let us try to continue our generalization by abstracting the type of the function arguments. A new statement and a new proof are obtained :

```
Lemma f2_permute :
∀E : Set.
 ∀f : E→E→E.
  ∀f_assoc_l : ∀n, m, p : E.(f n (f m p)) = (f (f n m) p).
   ∀f_sym : ∀n, m : E.(f n m) = (f m n)).
    ∀n, m, p : E.((f n (f m p)) = (f m (f n p))).
Proof
λE : Set.
  λf : E→E→E.
   λf_assoc_l : ∀n0, m0, p0 : E.(f n0 (f m0 p0)) = (f (f n0 m0) p0).
     λf_sym : forall n0, m0 : E.(f n0 m0) = (f m0 n0).
       λn, m, p : E.
         (eq_ind_r E (f (f m n) p) λn0 : E.(f n (f m p)) = n0
                   (eq_ind_r E (f n m) λ.n0 : E.(f n (f m p)) = (f n0 p)
                   (f_assoc_l n m p) (f m n) (f_sym m n)) (f m (f n p))
                  (f_assoc_l m n p))
```

5.1 Instantiations :

Trivialy, this generalization can be instantiated with the multiplication on natural numbers but we can also instantiate the theorem with other functions such as the addition on multi-variables monomials as defined by Pottier[5]. We get :

```
Lemma mon_permute :
 ∀k : nat.∀n, m, p : (mon k).
  ((mult_mon k n (mult_mon k m p)) = (mult_mon k m (mult_mon k n p))).

Proof λk : nat.(f2_permute (mon k)
   (λi : (mon k).λj : (mon k).(mult_mon k i j ))
   (λi : (mon k).λj : (mon k).λl : (mon k).(mult_mon_assoc k i j l))
   (λi : (mon k).λj : (mon k).(mult_mon_com k i j))).
```

6 Limitations

We could be tempted to push the abstraction further. Indeed, why limit the statement to the equality relation *eq*. The only property of this relation which

[5] (http://www-sop.inria.fr/croap/CFC/buch/Monomials.html)

is used is the rewriting principle :

$$eq\_ind\_r : \forall A : Set.\forall x : A.\forall P : A \to Prop.(Px) \to \forall y : A.y = x \to (Py)$$

The statement thus remains true for any relation R verifying this principle. Thus it is possible to get the following generalization:

```
Lemma  f3_permute  :
  ∀E : Set.∀f : E→E→E).
   ∀R : ∀A : Set.A→A→Prop.
    ∀R_ind_r : ∀A : Set.∀x : A.∀P : A→Prop.(P x)→∀y : A.(R A y x)→(P y))
      ∀f_assoc_l  : ∀n, m, p : E.(R E (f n (f m p)) (f (f n m) p)).
        ∀f_sym  : ∀n, m : E.(R E (f n m) (f m n)).
          ∀n, m, p : E.(R E(f n (f m p)) (f m (f n p))).
```

which can be proved by the term below:

```
λE : Set.
  λf : E→E→E.
     λR : (A : Set)A→A→Prop.
      λR_ind_r : ∀A : Set.∀x : A.∀P : A→Prop.(P x)→∀y : A.(R A y x)→(P y).
       λf_assoc_l : ∀n0, m0, p0 : E.(R E (f n0 (f m0 p0)) (f (f n0 m0) p0)).
        λf_sym : ∀n0, m0 : E.(R E (f n0 m0) (f m0 n0)).
         λn, m, p : E.
         (R_ind_r E (f (f m n) p)λn0 : E.(R E (f n (f m p)) n0)
           (R_ind_r E (f n m) λn0 : E.(R E (f n (f m p)) (f n0 p))
             (f_assoc_l n m p) (f m n)) (f_sym m n)) (f m (f n p))
           (f_assoc_l m n p)).
```

6.1 Instantiations:

We still check this generalized theorem by instantiating it with the multiplication on *nat*:

```
Definition mult_permute  :=
 λn, m, p : nat.(f3_permute
          nat  plus eq eq_ind_r plus_assoc_l plus_sym) :
   ∀n, m, p : nat.((plus n (plus m p)) = (plus m (plus n p))).
```

But eq_ind_r characterizes the equality of Leibniz, and we will not be able to re-use this lemma with other definite equalities.

Let us consider for example the polynomials such as they were defined by Pottier and Théry[Thé98]. They are represented by the table of their coefficients. The addition of two polynomials is obtained by summing their coefficients of same degree. An equality relation eqP is defined to identify for example 0 and $0 + (0 * x)$ which are different for the equality eq.

Thus, we tried to prove the *permute* property for the addition of these polynomials, that is to say the following lemma:

```
Lemma plusP_permute :
  ∀n, m, p : P.(eqP (plusP n (plusP m p)) (plusP m (plusP n p))).
```

But it is not possible to reuse our proof term. We can try to establish a generalization of eq_ind_r such as for example in the case of the polynomials with integer coefficients.

```
Lemma eq_ind_r_int :
∀x : (P nat).∀ P1 : (P nat)→Prop.
  (P1 x)→∀y : (P nat).(eqP nat (eq nat) λx : nat.x = O  y x)→(P1 y)
```

But this result is not true, otherwise taking $\lambda z : (Pnat).x = z)$ as $P1$ we could prove that eqP imply eq, which is not true.

We can nevertheless complete the proof of $plusP\_permute$ by the following script:

```
Intros.
(Apply eqP_trans with (plusP (plusP m n) p); Auto).
(Apply eqP_trans with (plusP (plusP n m) p); Auto).
```

Its proof term, given below, has nothing to do with the one we previously had:

```
∀n, m, p : P.
        (eqP_trans (plusP n (plusP m p)) (plusP (plusP m n) p)
          (plusP m (plusP n p))
          (eqP_trans (plusP n (plusP m p)) (plusP (plusP n m) p)
            (plusP (plusP m n) p) (plusP_associative n m p)
            (eqP_sym (plusP (plusP m n) p) (plusP (plusP n m) p)
              (eqP_sym (plusP (plusP n m) p) (plusP (plusP m n) p)
                (eqP_sym (plusP (plusP m n) p) (plusP (plusP n m) p)
                  (plusP_compl (plusP m n) (plusP n m) p
                    (plusP_commutative m n))))))
          (eqP_sym (plusP m (plusP n p)) (plusP (plusP m n) p)
            (plusP_associative m n p)))
```

Thus, this example shows that it is not always desirable to generalize as much as possible (i.e. to abstract all the free identifiers appearing in a statement).

7 Abstracting theories

Generalisation can be represented in a more structural way. We have said that the permutation property remains true for any binary operator f, which is associtive and commutative. Such structure is a commutative semi-group and may be represented by a record.

```
Record semi_group : Type :=
  {sg_s : Set;
   sg_law : sg_s → sg_s → sg_s;
   sg_assoc : ∀x, y, z : sg_s.(sg_law x (sg_law y z)) = (sg_law (sg_law x y) z)}.

Record semi_group_com : Type :=
  {sg :> semi_group;
   sg_com : ∀x, y : (sg_s sg).(sg_law sg x y) = (sg_law sg y x)}.
```

The notation :> is a coercion facility that allows to see a commutative semi-group as a semi-group.

Now we can build a lemme expressing the permute property for the internal law of a commutative semi-group.

```
Lemma sg_permute : ∀sg1 : semi_group_com.∀n, m, p : (sg_s sg1).
   ((sg_law sg1 n (sg_law sg1 m p)) = (sg_law sg1 m (sg_law sg1 n p))).

Proof
λsg1 : semi_group_com.
λn, m, p : (sg_s sg1).
 (eq_ind_r (sg_s sg1) (sg_law sg1 (sg_law sg1 m n) p)
       λn0 : (sg_s sg1).(sg_law sg1 n (sg_law sg1 m p)) = n0
       (eq_ind_r (sg_s sg1) (sg_law sg1 n m)
             λn0 : (sg_s sg1).
              (sg_law sg1 n (sg_law sg1 m p)) = (sg_law sg1 n0 p)
             (sg_assoc sg1 n m p) (sg_law sg1 m n) (sg_com sg1 m n))
       (sg_law sg1 m (sg_law sg1 n p)) (sg_assoc sg1 m n p)).
```

In this generalization of our initial *permute* lemma, all abstracted identifier (ie the Set, the operator and the properties) have been grouped in one that denote a commutative semi-group. The internal law and theirs properties have respectively been substituted to the operator and the operator properties.

As usual, we check that this generalization specialized has show below for addition on natural numbers :

```
Definition sg_com_nat :=
 (Build_semi_group_com
   (Build_semi_group nat plus plus_assoc_l)
    plus_sym).

Lemma plus_permute :
∀ n, m, p : nat.((plus n (plus m p)) = (plus m (plus n p))).
Exact (sg_permute sg_com_nat).
Save.
```

8 Remarks on the abstraction of inductively defined objects

In the previous sections we have shown several exemple of abstation of inductively defined objects. In fact we can distinguish two cases:

I The identifiers associated with the constructors of the abstracted object do not appear in the term to be generalized. We say that this is a "silent object" in the term. In such case, we have just to abstract this identifier as we have done to build $f2\_permut$.

II The Identifiers associated with the constructors of the abstracted object appear in the term to be generalized. Here we can distinguish two sub-cases

II-i If they do not appear in a *case*-expression, then they have just to be abstracted. The example below shows how to abstrat nat in the dependent type Tab. $(Tab\ n)$ is the type of the lists of length n.

$$
\begin{array}{l}
Inductive\ Tab\ \lambda s : Set.\ :\ nat{\rightarrow}Set\ :=\\
\quad Tnil\ :\ (Tab\ s\ O)\\
\quad |\ Tcons\ :\ \forall n : nat.s{\rightarrow}(Tab\ s\ n){\rightarrow}(Tab\ s\ (S\ n)).\\
\\
Inductive\ TabAbs\ \lambda s : Set.\lambda s1 : Set.\lambda.OAbs : s1\lambda.SAbs : s1{\rightarrow}s1.\ :\ s1{\rightarrow}Set\ :=\\
\quad TnilAbs\ :\ (TabAbs\ s\ s1\ OAbs\ SAbs\ OAbs)\\
\quad |\ TconsAbs\ :\ (n : s1)s{\rightarrow}(TabAbs\ s\ s1\ OAbs\ SAbs\ n){\rightarrow}\\
\qquad\qquad (TabAbs\ s\ \ s1\ OAbs\ SAbs\ (SAbs\ n)).
\end{array}
$$

In $TabAbs$ the dependant parameter nat, and its two constructors 0 and S have been abstracted.

II-ii If they appear in a case expression. This happens for example when abstracting inductive defined type in function definition. As example let us see what happens is we try to abstrat nat in the definition of $plus$. In fact here also there is two possibility :

- If the definition of $plus$ is done using $fixpoint$ and $case$ we try to translate it in a definition using a recursor.
- Else, if it is still defined with a recusor everything is good.

$$
\begin{array}{l}
Definition\ \ plusl\ :=\\
\lambda H : nat.\\
\ (nat\_rec\ \lambda\_ : nat.nat{\rightarrow}nat\ \ \lambda m : nat.m\\
\quad \lambda\_ : nat.\lambda H0 : (nat{\rightarrow}nat).\lambda H1 : nat.(S\ \ (H0\ \ H1))\ \ H).
\end{array}
$$

- After getting a function definition with recusor, as before, the transformation is syntactic. Abstacting, the type indentifier, the constructors and the recursors, we get :

$$
\begin{array}{l}
Definition\ \ plusAbs\ :=\\
\lambda s1.\lambda Set.\lambda OAbs : s1.\lambda SAbs : s1{\to}s1.\\
\ \lambda s1\_rec : (\forall P : (s1{\to}Set).\\
\qquad\qquad (P\ \ (OAbs)){\to}((n : s1)(P\ \ n){\to}(P\ \ (SAbs\ \ n))){\to}(n : s1)(P\ \ n)).\\
\quad \lambda\ \ H : s1.\\
\qquad (s1\_rec\ \ \lambda\_ : s1.s1{\to}s1\ \ \lambda m : s1.m\\
\qquad\quad \lambda\_ : s1.\lambda H0 : (s1{\to}s1).\lambda H1 : s1.(SAbs\ \ (H0\ \ H1))\ \ H).
\end{array}
$$

The example of $f3\_permute$ showed that even possible such transformation are note alway useful.

In fact, there is one case where they are useful, This is when we want to switch from a data representation to another one. As example this this allows to switch from the usual unary representation of natural number to the dyadic one defined by[6] : $Inductive\ bin : Set := BH : bin | BO : bin \to bin | BI : bin \to bin.$

Now In order to re-instantiate $TabbAbs$, $plusl$ or another abstraction on nat, we have just to define the successor and the unary recursor for bin. This is :

```
Definition Sbin [b:bin] : bin :=
Cases b of
BH =>(BO BH)
|(BO x) =>(BI x)
|(BI x)=>(BO (Sbin x))
end.
Lemma bin_rec :(P:(bin->Set))(P (BH))->((n:bin)(P n)->(P (Sbin n)))->(n:bin)(P n).
```

Those problems are discussed in detail in [BP01,MB01].

9 Conclusion and perspective

We have suggested a practical generalization mechanism and detailed a number of situation in which such a mechanism may be used. We also highlighted the problems and difficulties that appear in practical use and propose pragmatic solutions.

We saw that in an existing developments many proofs use only some properties of the "objects" on which they work. The relations between objects may be highlighted once for all by computing a dependency graph[PBR98] allowing to build a more abstract theory. The certification of mathematical algorithms for computer algebra must be based on a certified implementation of the basic mathematical structures (groups, rings, algebra etc). In the long run, generalization should make it possible to recover for free, all the theorems in the existing developments which are still valid for a new structure. As example in the case of groups, proofs which have been done for an instance of these theories such as integer or polynomials may be recovered. In relation to the introduction of the concept of modules in proof assistant, this could make it possible to develop parameterized theories starting from specific instances. This idea of structuring "a

[6] here, intuitively $BH = O$, $BO\ y = 2y + 1$ and $BII\ y = 2y + 2$.

posteriori", is similar to Siff and Reps's works[SR96] in the field of programming languages, when they try to generate generic *C++* code starting from C code.

References

[Aa99] D. Aspinall and al. Proof general 3.0, December 1999. http://zermelo.dcs.ed.ac.uk/p̃roofgen.

[Ba96] Janet Bertot and al. *User Guide to the CtCoq Proof Environment.* INRIA, Feb 1996.

[Ba99] B. Barras and al. *The Coq Proof Assistant Reference Manual.* INRIA, December 1999. Version 6.3.1.

[Bou00] S. Boulme. *Spécification d'un environnement dedié à la programmation certifiée de bibliothèques de Calcul Formel.* PhD thesis, Université Paris 6, décembre 2000.

[BP01] G. Barthe and O. Pons. Type Isomorphisms and Proof Reuse in Dependent Type Theory. In F. Honsell and M. Miculan, editors, *Proceedings of FOSSACS'01*, volume 2030, pages 57–71, 2001.

[Ca86] R. Constable and al. *Implementing mathematics with the Nuprl proof development system.* Prentice-Hall, 1986.

[FH94] A. Felty and D. Howe. Generalization and reuse of tactic proofs. *Lecture Notes in Computer Science*, 822:1–15, 1994.

[HR92] R. Hasker and U. Reddy. Generalization at higher types. In D. Miller, editor, *Proceedings of the Workshop on the λProlog Programming Language*, pages 257–271, Philadelphia, Pennsylvania, July 1992. University of Pennsylvania. Available as Technical Report MS-CIS-92-86.

[KW94] T. Kolbe and C. Walther. Reusing proofs. In *Proc. of the 11th ECAI*, pages 80–84, Amsterdam, The Netherlands, 1994.

[LEG] The LEGO World Wide Web page. url http://www.dcs.ed.ac.uk/home/lego.

[MB01] N. Magaud and Y. Bertot. Changement de représentation des structures de données en Coq: le cas des entiers naturels. In *JFLA'2001*, 2001.

[PBR98] O. Pons, Y. Bertot, and L. Rideau. Notions of dependency in proof assistants. In *Electronic Proceedings of "User Interfaces for Theorem Provers 1998 "*, Sophia-Antipolis, France, 1998.

[Pfe91] F. Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[Plo69] G. D. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163, Edinburgh, 1969. Edinburgh University Press.

[RS93] W. Reif and K. Stenzel. Reuse of proofs in software verification. *Foundations of Software Technology and Theoretical Computer Science*, 13, 1993.

[SR96] M. Siff and T. Reps. Program generalization for software reuse: From C to C++. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 21_6 of *ACM Software Engineering Notes*, pages 135–146, New York, October 16–18 1996. ACM Press.

[Thé98] L. Théry. A certified version of Buchberger's algorithm. In *Automated Deduction (CADE-15)*, volume 1421 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, July 1998.