Changing Data Structures in Type Theory: A Study of Natural Numbers

Nicolas Magaud and Yves Bertot

INRIA Sophia Antipolis

Abstract. In type-theory based proof systems that provide inductive structures, computation tools are automatically associated to inductive definitions. Choosing a particular representation for a given concept has a strong influence on proof structure. We propose a method to make the change from one representation to another easier, by systematically translating proofs from one context to another. We show how this method works by using it on natural numbers, for which a unary representation (based on Peano axioms) and a binary representation are available. This method leads to an automatic translation tool that we have implemented in *Coq* and successfully applied to several arithmetical theorems.

1 Introduction

Mechanical theorem provers can be used to develop provably correct software and formalize large bodies of mathematics. Proofs for mathematical objectives are eventually very useful for proofs of software components, when the correctness of algorithms rely on arbitrarily complex mathematical notions, but very often the proof styles are different. For plain mathematics, simplicity of concepts is of paramount importance. For software proofs, efficiency plays a more important role.

When considering natural numbers, the difference of style between mathematical proofs and software proofs is embodied in two different representations of numbers. With the *unary* representation, natural numbers are described as an inductive set with two constructors: 0 and the successor function. The simplicity is perfect, inductive proofs only have two cases, and this representation is preferred in most proof developments about the mathematical properties of natural numbers. But the representation of a number has a size proportional to the number itself. Computers use a more compact representation, called the binary representation. A natural number is either 0, or a sequence of *bits*, ones and zeros, that starts with a *one*. Inductive proofs on this structure naturally have four cases: two base cases corresponding to 0 and 1 and two other cases corresponding to numbers of the form $2 \times x$ and $2 \times x + 1$ when x is already a number different from 0.

Up until now, a large number of results have been established about natural numbers, where the proofs make an intensive use of the unary representation of natural numbers. However, it may sometimes be desirable to switch to the more efficient binary representation. Should all results be proven again, or is there a way to re-use all the proofs that have already been done, either by translating them from one data-structure to another, or by establishing these results for an abstract data type that can be shown to be "made concrete" in both representations?

This study concentrates on natural numbers, but it should apply to many other contexts where initial choices of concrete data types might need to be re-considered as software evolves.

2 Proof Representation

In type-theory based logical systems, the Curry-Howard isomorphism establishes a correspondence between logic and typed λ-calculus, where types are logical formulas and λ-terms are proofs. This isomorphism makes it possible to identify the implication with a function type. Moreover, universal quantification is identified with a general notion of *dependent* function types, where the type for the returned value depends on the input value.

For instance, let us consider a proof of "transitivity of implication" for propositions P, Q, and R. The statement has the following shape:

$$(R \Rightarrow P) \Rightarrow (P \Rightarrow Q) \Rightarrow R \Rightarrow Q. \tag{1}$$

The proof for this statement is represented by the following term:

$$\lambda th_1 : (R \Rightarrow P); th_2 : (P \Rightarrow Q); th_3 : R.(th_2\ (th_1\ th_3)). \tag{2}$$

According to Heyting-Kolmogorov's semantics, we can also associate a computational interpretation to proofs. We can consider the term (2) as a function, which constructs a proof of Q when given proofs th_1, th_2, th_3, respectively for $R \Rightarrow P$, $P \Rightarrow Q$, and R. Just respecting the type discipline, we know that applying th_1 to th_3 is well-formed and returns a term of type P (i.e., a proof of P), then applying th_2 finishes the proof.

Proof Translation: If P', Q', and R' are the same properties as P, Q, and R, but now expressed with respect to a new concrete representation, it is easy to reuse the proof (2) to obtain the following statement:

$$(R' \Rightarrow P') \Rightarrow (P' \Rightarrow Q') \Rightarrow R' \Rightarrow Q' \tag{3}$$

A proof of this statement then becomes:

$$\lambda th'_1 : (R' \Rightarrow P'); th'_2 : (P' \Rightarrow Q'); th'_3 : R'.(th'_2\ (th'_1\ th'_3)). \tag{4}$$

Application to Induction Principles: In practice, when one chooses a data type, proof tools automatically generate induction principles. As a result, the proof structures are influenced by the chosen data types and some theorems available in one formalization will be missing in the other one. One important part of the translation work is to recover theorems corresponding to induction principles from the old setting in the new setting.

Convertibility: Reasoning about inductive types also involves computing recursively over these types. In type-theory based proof systems, the tradition is to include computation in the conversion mechanisms that are automatically used by the type-checker to compare two terms. Because the recursion mechanism is intimately linked with the structure of the data types, this has a tremendous impact on the translation of proofs: terms that are convertible in one setting may not be convertible in the new setting. If P_A and P_B are considered equivalent in the initial setting, the following term represents a correct proof:

$$\lambda th_1 : (P_A \Rightarrow Q); th_2 : (R \Rightarrow P_B); th_3 : R.(th_1\ (th_2\ th_3)). \tag{5}$$

The statement being proven is the following one:

$$(P_A \Rightarrow Q) \Rightarrow (R \Rightarrow P_B) \Rightarrow R \Rightarrow Q. \tag{6}$$

When changing over to a new data-structure, P'_A, and P'_B, obtained by translating P_A and P_B respectively may not be convertible anymore. It then becomes necessary to modify the structure of the proof term (5) and replace the implicit equivalence between P_A and P_B by the explicit application of a theorem $th'_4 : P'_B \Rightarrow P'_A$ that we will have to prove on the side. In this case, the translation of (6) becomes the following term:

$$\lambda th'_1 : (P'_A \Rightarrow Q'); th'_2 : (R' \Rightarrow P'_B); th'_3 : R'.(th'_1\ (th'_4\ (th'_2\ th'_3))).$$

In this paper, we describe the practical problems we have encountered in an attempt to transform theorems from the unary representation to the binary representation. We first recall the basics of inductive data type descriptions in *Coq* (section 3), we then show how to make the conversion steps in the proof explicit (section 4). In a third stage, we study how to represent functions and their computational behavior (section 5). We finally present the tools we need to change from one inductive data type to another, focusing on natural numbers and their unary and binary representations (section 6). We then present an overview of related work and examine the perspectives of this work.

3 Inductive Types in *Coq*

3.1 General Characteristics of *Coq*

The *Coq* system provides an implementation of the *Calculus of Inductive Constructions* [3, Chap.4]. It is a typed λ-calculus with dependent types [16] with capabilities for inductive definitions [14]. *Coq* provides an interactive mode for developing proofs by backward chaining, with a collection of proof commands called tactics, that decompose goal logical formulas into simpler sub-goals. Each tactic makes it possible to perform elementary reasoning steps. Eventually, all proof steps are combined in a λ-term where the reasoning steps are represented using λ-calculus constructions.

Inductive types are defined by giving a collection of functions whose codomain is the type being defined and whose arguments may be in the inductive type being defined (in fact, there are precise constraints on how the inductive type may be used in the arguments but we will use only a simple form of inductive types here). These functions are called constructors. For instance, unary natural numbers are defined in the following manner:

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

The inductive type is called `nat`, the functions given to define it are `O` (a function with no argument, i.e., an element in the type) and `S` (a unary function). The automatic tools in *Coq* use this definition to automatically construct a *structural induction principle* and recursion operator that makes it possible to define recursive functions. The induction principle has the following shape:

```
nat_ind
 : (P:(nat->Prop))(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
```

In the calculus of inductive constructions, the recursion operator associated to an inductive type is integrated in the conversion mechanisms that make it possible to compare terms. It can be assimilated with a term `nat_rec` with the following type:

```
nat_rec
 : (P: (nat -> Set))(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
```

The reduction rules associated with this recursion operator are as follows:

$$(\texttt{nat\_rec}\ P\ v_0\ f\ O) \rightarrow v_0 \quad (\texttt{nat\_rec}\ P\ v_0\ f\ (S\ n)) \rightarrow (f\ n\ (\texttt{nat\_rec}\ P\ v_0\ f\ n))$$

The definition of recursive functions is not always translated in terms of the recursion operators. Instead, the definition of a recursive function can be viewed as giving rise directly to reduction rules. For instance the factorial function is defined as follows:

```
    Fixpoint fact [n:nat] : nat :=
        Cases n of O => (S O)
                 | (S p) => (mult (fact p) (S p))
    end.
```

Corresponding reduction rules are:

$$(\texttt{fact}\ 0) \rightarrow (S\ 0) \qquad (\texttt{fact}\ (S\ n)) \rightarrow (\texttt{mult}\ (\texttt{fact}\ n)\ (S\ n))$$

The logical system does not distinguish between convertible terms. Put in other terms, $(\texttt{fact}\ 0)$ and $(S\ 0)$ are identified. As a result, convertibility may hide some parts of the reasoning steps, a particularity that has been thoroughly exploited in *reflexion* or *two-level* approaches [8,4].

4 Making Conversions Explicit

Some reasoning steps may be missing in proof terms because of the convertibility rules which may identify syntactically different terms. We start by describing through an example what implicit computational steps we would like to make explicit in a proof term. We then present the method developed to achieve this goal and describe an algorithm. We conclude by showing what happens when the algorithm is applied to an example.

4.1 An Example

Throughout this section, we consider the theorem $\mathsf{plus\_n\_O}$.

$$\forall n \in \mathsf{nat} \quad n = (\mathsf{plus}\ n\ \mathsf{O}) \tag{7}$$

A proof (as a λ-term) of this property is :

$$\begin{aligned}\lambda n : \mathsf{nat}.(\mathsf{nat\_ind}\ &(\lambda n0 : \mathsf{nat}.n0 = (\mathsf{plus}\ n0\ \mathsf{O}))\\ &(\mathsf{refl\_equal\ nat\ O})\\ &\lambda n0 : \mathsf{nat}; H : (n0 = (\mathsf{plus}\ n0\ \mathsf{O})).\\ &\quad(\mathsf{f\_equal\ nat\ nat\ S}\ n0\ (\mathsf{plus}\ n0\ \mathsf{O})\ H)\ n)\end{aligned}$$

It proceeds by induction on n, through the principle called $\mathsf{nat\_ind}$.
The term $(\mathsf{refl\_equal\ nat\ O})$ is a proof that $\mathsf{O} = \mathsf{O}$. However, thanks to convertibility rules, it is also a proof of $\mathsf{O} = (\mathsf{plus\ O\ O})$. In the same way,

$$\lambda n0 : \mathsf{nat}; H : (n0 = (\mathsf{plus}\ n0\ \mathsf{O})).\ (\mathsf{f\_equal\ nat\ nat\ S}\ n0\ (\mathsf{plus}\ n0\ \mathsf{O})\ H)$$

is a proof of

$$\forall n0 : \mathsf{nat} \quad n0 = (\mathsf{plus}\ n0\ \mathsf{O}) \Rightarrow (\mathsf{S}\ n0) = (\mathsf{S}\ (\mathsf{plus}\ n0\ \mathsf{O}))$$

It is also a proof of $\forall n0 : \mathsf{nat} \quad n0 = (\mathsf{plus}\ n0\ \mathsf{O}) \Rightarrow (\mathsf{S}\ n0) = (\mathsf{plus}\ (\mathsf{S}\ n0)\ \mathsf{O})$ which is the expected type for the third argument of the induction principle $\mathsf{nat\_ind}$.

4.2 Computing Expected and Proposed Types

The first step in our work is to extract the implicit computational steps from the proof terms. To this end, we use a technique derived from Yann Coscoy's work [5] on proof explanation. This technique is also very close to the methods for type-checking dependent types in the original ALF system [11]. We need to determine positions within the proof term where expected and proposed types differ. We restrict our work to concrete datatypes and concrete equalities and will not be able to work with an abstract algebraic "book" equality. Here we focus on Leibnitz's equality which is the basic equality of Coq.

4.3 Expected and Proposed Types

The algorithm we present in the next section uses intensively the notions of expected and proposed type for a term. Given a term t, its expected type T_E is the type required for the whole term (whose t belongs to) to be well-typed. Its proposed type T_P is the one inferred by the type system.
Let us consider the example of an application $(t_1\ t_2)$. If the types inferred by the system are the following ones:

$$t_1\ :\ T\ \rightarrow\ U \qquad \& \qquad t_2\ :\ T'$$

Then the expected type for t_2 is the argument T of the functional type $T\ \rightarrow\ U$, whereas its proposed type is T'. The application $(t_1\ t_2)$ is well-typed provided the expected and proposed types for t_2 are convertible. When using the formulas-as-types (and proofs-as-terms) analogy, we see that the expected type is a *proof obligation*, i.e. it states what we are required to prove. The proposed type is a *proof witness*, i.e. it can be viewed as the formula whose term (here t_2) is actually proven.

4.4 The Algorithm

The algorithm is designed to work on proof terms which are well-typed. Its aim is to locate positions where implicit ι-reduction steps occur in the proof term. It eventually builds a new proof term, with no more implicit (ι-reduction) steps. All these steps are recorded as logic variables that need to be proven on the side.

The terms we consider in this study are constants, variables, inductive types and their constructors, λ-abstractions, product types, and applications. The calculus of constructions also contains a case analysis operator (`Cases`) but it involves important complications and we treat this operator in a separate section.

The algorithm takes a statement T, its proof t, and a context C as input. It proceeds by recursive case analysis on t, and returns a new term and a list of conjectures.

- **Variables, Constants, Inductives Data Types, Constructors**
  All these constructs remain unchanged and are simply returned.
- **Abstraction** $t \equiv \lambda x : A.b$
  By hypothesis, the proof term we consider is well-typed. This ensures that T can be reduced (via weak-head normal form computation) to a product $\forall x : A'.B$. Moreover, we know that A and A' are convertible modulo $\beta\delta\iota$-reduction. We choose to add $x : A'$ in the context C.
  We then call the algorithm recursively with new input: b instead of t, B instead of T, and $x : A'\ ;\ C$ instead of C.
- **Application** $t \equiv (h\ u_1 \dots\ u_n)$
  Here, we only know that the expected type for the whole application is T. There is no expected type for the function h. However, we can compute its proposed type T_P. It is necessarily a product since the initial term is well-typed. We can infer the expected type TE_{u_1} for the first argument. We then

compare that type with TP_{u1}, that can be inferred by the system. If the types TE_{u_1} and TP_{u_1} are convertible by $\beta\delta$ conversion the algorithm simply stores an intermediary value $v_1 = u_1$. If these two types are not convertible by $\beta\delta$ conversion, then the algorithm constructs a new conjecture c_1 whose type is $\forall x_1 : t_1, \ldots x_k : t_k.TP_{u_1} \to TE_{u_1}$ where the variables x_i are the variables that appear free in TP_{u_1} and TE_{u_1} and whose type is given in C; it stores an intermediary value $v_1 = (c_1\ x_1 \cdots\ x_k\ u_1)$. This process is repeated for all variables u_i, each time creating an intermediary value v_i and, possibly, a conjecture c_i. In the end, the algorithm constructs a term $v = (h\ v_1 \ldots v_n)$. If this term v has type T, then the algorithm returns v and the new list of conjectures. If this term has a type T' different from T, then the algorithm constructs yet another conjecture c of type $\forall x_1 : t_1, \ldots x_k : t_k.T' \to T$ and returns the term $(c\, x_1 \cdots\ x_k\ (h\ v_1 \cdots v_n))$ and a list of conjectures containing c and all the c_i's.
In the initial setting, all the conjectures are simple consequences of the convertibility rule. In the target setting, these conjectures will need to be proven.

4.5 What Happens to Our Example ?

In our example, the proof term of `plus_n_O` is transformed into the following one:

$$\begin{array}{l}
\lambda n : \mathsf{nat}.(\mathsf{nat\_ind} \\
\qquad\qquad (\lambda n0 : \mathsf{nat}.n0 = (\mathsf{plus}\ n0\ \mathsf{O})) \\
\qquad\qquad (Ha\ (\mathsf{refl\_equal\ nat\ O})) \\
\qquad\qquad \lambda n0 : \mathsf{nat}; H : (n0 = (\mathsf{plus}\ n0\ \mathsf{O})). \\
\qquad\qquad\quad (Hb\ n0\ (\mathsf{f\_equal\ nat\ nat\ S}\ n0\ (\mathsf{plus}\ n0\ \mathsf{O})\ H))\ n)
\end{array}$$

Two conjectures Ha and Hb are generated in order to relate expected and proposed types in the branches of the application of the induction principle.

$$\begin{array}{rcl}
Ha : \mathsf{O} = \mathsf{O} & \Rightarrow & \mathsf{O} = (\mathsf{plus\ O\ O}) \\
Hb : \forall n : \mathsf{nat}.(\mathsf{S}\ n) = (\mathsf{S}\ (\mathsf{plus}\ n\ \mathsf{O})) & \Rightarrow & (\mathsf{S}\ n) = (\mathsf{plus}\ (\mathsf{S}\ n)\ \mathsf{O})
\end{array}$$

We see they can be proven easily, by first introducing the premises and then using the reflexivity of Leibniz's equality. This works because the terms on both sides of the equality are convertible modulo $\beta\delta\iota$-reduction.

5 Representing Functions

When translating proofs from one setting to the other, it is also necessary to find the corresponding representations for the functions that are used in the proofs. The corresponding representation for a given function may use a completely different algorithm, for instance to benefit from the characteristics of the new data structure. When the algorithm changes, we still need to express that the initial function and its representation in the new setting do represent the same function. This will be done by showing that the new function satisfies equalities that characterize the behavior of the initial function. We still need to show how to find these equalities.

We describe this on the example of addition. In the unary setting, addition works by moving as many `S`'s in front of the second argument as there are `S`'s in the first argument. This is expressed with a function that has this form:

```
Fixpoint plus[n:nat] : nat -> nat :=
   Cases n of O => [m:nat] m
     | (S p) => [m:nat](S (plus p m))
   end.
```

This notation is specific to *Coq*, and is very close to a `let rec` or `letrec` definition in ML.

As we already mentioned, such a definition adds to the convertibility rules in the proof system, so that $(\mathsf{plus}\ (\mathsf{S}\ n)\ m)$ and $(\mathsf{S}\ (\mathsf{plus}\ n\ m))$ actually are convertible. In some theorems, the first term may be provided when the second term is requested. If we work in the binary setting, it is most likely that this convertibility will not hold anymore. On the other hand, it is possible to prove the equality that expresses that both terms are equal.

Generating Equalities: From a given function definition, it is possible to derive a collection of equalities that express the actual convertibility rules added in the system. These equalities are constructed by an analysis on the text of the definition. A first step to producing these equalities is to isolate the fixpoint equality as already studied in [2]. For the `plus` example, this equality has the following form.

$$\forall n : \mathsf{nat}.\ \ (\mathsf{plus}\ n) = \begin{array}{lll} \mathtt{Cases}\ n\ \mathtt{of} & & \\ \ \ 0 & \texttt{=>} & [m{:}\mathsf{nat}]m \\ |(\mathsf{S}\ p) & \texttt{=>} & [m{:}\mathsf{nat}](\mathsf{S}\ (\mathsf{plus}\ p\ m) \\ \mathtt{end} & & \end{array}$$

However, this equality is still very linked to the unary data structure, because it makes use of the `Cases` construct. For this `Cases` construct to be valid, it is necessary to work in a setting where O and S are data type constructors. To abstract away from the data structure, we exhibit equalities corresponding to each conversion rule in the `Cases` construct. In our example the function performs two different tasks, depending on whether n is 0 or $(\mathsf{S}\ p)$ for some p.

If n is 0, then the left-hand side of the equality becomes $(\mathsf{plus}\ 0)$. The right-hand side is simplified according to corresponding rule in the `Cases` construct. The equality becomes as follows:

$$(\mathsf{plus}\ 0) = [m : \mathsf{nat}]m$$

If n is $(\mathsf{S}\ p)$, the equality becomes:

$$\forall p : \mathsf{nat}.\ (\mathsf{plus}\ (\mathsf{S}\ p)) = [m : \mathsf{nat}](\mathsf{S}\ (\mathsf{plus}\ p\ m))$$

In fact, since type-theory based proof systems usually do not compare functions extensionally, it is better to modify these equalities to avoid comparing functions, but only values of these functions when these values do not have a function type. The equalities thus become as follows:

$$\forall m : \mathsf{nat}.\ (\mathsf{plus}\ 0\ m) = m$$
$$\forall p, m : \mathsf{nat}.\ (\mathsf{plus}\ (\mathtt{S}\ p)\ m) = (\mathtt{S}\ (\mathsf{plus}\ p\ m))$$

Now, if we want to translate a proof about addition in the unary setting to another setting, we need to prove these equations for the new addition. These equations can then be used to simulate convertibility.

6 Mapping Binary Representation to the Unary Setting

The binary representation of natural numbers is inspired from the representation of integer numbers proposed by P. Crégut as an implementation basis for the `Omega` tactic, one of the most useful decision procedures for numerical problems provided in *Coq*.

6.1 Constructing the Bijections

The set of strictly positive numbers is the set generated by 1 and the two functions $x \mapsto 2 \times x$ and $x \mapsto 2 \times x + 1$. This set is constructed using the following inductive definition:

```
Inductive positive : Set :=
  one: positive                (* 1 *)
| pI: positive ->positive      (* 2x+1, x>0 *)
| pO: positive ->positive.     (* 2x, x>0 *)
```

The whole set of inductive numbers, including 0 is then described as the disjoint sum of $\{0\}$ and strictly positive numbers.

```
Inductive bin : Set := zero : bin | pos : positive -> bin.
```

From these definitions, the structural induction principles that are generated basically express the following statement:

$$P\ (0) \wedge P\ (1) \wedge (\forall p \in \mathsf{positive}\ .\ P\ (p) \Rightarrow P\ (2p{+}1)) \wedge (\forall p \in \mathsf{positive}\ .\ P\ (p) \Rightarrow P\ (2p))$$
$$\Rightarrow\ \forall n \in \mathsf{bin}\ .\ P\ (n)$$

Seen as sets, types bin and nat are isomorphic, but their induction principles establish a strong distinction between them. To reduce this distinction, we need to exhibit a function $\mathsf{S'}$, defined in the binary setting, representing the successor function. Our definition of $\mathsf{S'}$ relies on an auxiliary definition for strictly positive numbers, $\mathsf{aux\_S'}$:

```
Fixpoint aux_S' [n:positive] : positive :=
 Cases n of
  one (* 1 *)            => (pO one)        (* 2*1 *)
| (pO t) (* 2*t t>0 *)   => (pI t)          (* 2*t+1 *)
| (pI t) (* 2*t+1 t>0 *) => (pO (aux_S' t)) (* 2*(t+1) *)
 end.

Definition S' : bin -> bin :=
 [n:bin] Cases n of zero => (pos one) |
                            (pos u) => (pos (aux_S' u)) end.
```

We can now construct two functions from elements of one data type to the other:

$$\mathsf{BtoN} : \mathsf{bin} \rightarrow \mathsf{nat} \qquad \mathsf{NtoB} : \mathsf{nat} \rightarrow \mathsf{bin}.$$

The function BtoN also relies on an auxiliary definition that takes care of strictly positive numbers.

```
Fixpoint aux_BtoN [x:positive] : nat :=
Cases x of one    => (S O)
         | (pO t) => (mult (S (S O)) (aux_BtoN t))
         | (pI t) => (S (mult (S (S O)) (aux_BtoN t)))
end.

Definition BtoN :=
 [x:bin] Cases x of zero => O | (pos p) => (aux_BtoN p) end.
```

On the other hand, defining NtoB is a simple matter of repeating S′ the right number of times:

```
Fixpoint NtoB [n:nat] : bin :=
  Cases n of O => zero | (S x) => (S' (NtoB x)) end.
```

Then, we need to express that these two functions are the inverse of each other, or at least that NtoB is the left inverse of BtoN.

$$\mathsf{NtoB\_inverse} : \forall a \in \mathsf{bin} \quad \mathsf{NtoB}(\mathsf{BtoN}(a)) = a$$

This proof relies on an auxiliary theorem NtoB_mult2. This theorem re-phrases the interpretation of pO as multiplication by 2 in the unary setting.

6.2 Mapping the Induction Principle

With the equality NtoB_inverse, proving an induction principle for the binary setting that has the same shape as the unary setting induction principle is easy. This theorem we name new_bin_ind has the following statement:

$$\forall P : \mathsf{bin} \rightarrow Prop.(P\ \mathsf{zero}) \wedge (\forall x : \mathsf{bin}.P(x) \Rightarrow (P\ (\mathsf{S'}\ x))) \Rightarrow \forall x : \mathsf{bin}.P(x)$$

To prove this, we take an arbitrary x in bin and hypotheses $P(zero)$ and

$$\forall b : \mathsf{bin} \quad P(b) \Rightarrow P(S'(b))$$

(let us call this hypothesis h) and we replace x with $\mathsf{NtoB}(\mathsf{BtoN}(x))$. We then prove the result by induction over $\mathsf{BtoN}(x)$. The base case is $P(\mathsf{NtoB}(\mathsf{0}))$, which is equivalent to $P(\mathsf{zero})$, while the step case requires $P(\mathsf{NtoB}(\mathsf{S}(n)))$ knowing that $P(\mathsf{NtoB}(n))$ holds. Here $\mathsf{NtoB}(\mathsf{S}(n))$ is the same as $\mathsf{S}'(\mathsf{NtoB}(n))$ by the definition of NtoB and we can use the hypothesis h.

All the inductive reasoning steps in the unary representation can then easily be translated to the binary setting by relying on this new induction principle. In this sense, we have made it possible to abstract away from the actual structure of the data type: proofs done in the binary setting can have a single base case and a single recursive case, exactly like the proofs done in the unary setting.

6.3 Mapping Recursive Functions

In this section, we work on the example of addition. The purpose of changing to a binary representation of numbers is to use efficient algorithms for the basic operations like addition. For this reason, addition is described by a very different algorithm - the well-known carry adder - that happens to be structural recursive in the binary data structure.

To show that this addition algorithm represents faithfully the addition function as implemented in the unary setting we only need to show that it satisfies the characteristic equations of unary addition, as translated in the binary setting. Naming Bplus this binary addition, the equations are:

$$\begin{array}{ll} \mathsf{Bplus\_S}: & \forall p, q : \mathsf{bin}\ \mathsf{Bplus}(\mathsf{S}'(p), q) = \mathsf{S}'(\mathsf{Bplus}(p, q)) \\ \mathsf{Bplus\_O}: & \forall p : \mathsf{bin}\ \ \mathsf{Bplus}(\mathsf{zero}, p) = p \end{array}$$

The first equation is proven by induction on the binary structure, actually following the recursive structure of the function Bplus. Proving the second equation is very easy, since there is no recursion involved in this base case. However, since the addition function in the binary setting has been provided manually, this proof may require user-guidance.

The equations $\mathsf{Bplus\_S}$ and $\mathsf{Bplus\_0}$ can be added to the database of an automatic rewriting procedure that will automatically repeat rewriting with these equations (oriented left to right). Termination is ensured by the fact that termination of ι-reduction was already ensured for addition in the unary setting.

6.4 Proving Conjectures

In an earlier stage of our work (section 4), we make ι-conversions occurring in proofs explicit. The result of this work is a new presentation of proofs accompanied with a collection of conjectures that correspond to the reasoning steps that are performed by ι-conversion in the initial proof. The next stage of our work is to prove translations of these conjectures in the new setting.

In our example on the proof of plus_n_O, this yields two conjectures:

$$\mathsf{Ha'} : \mathsf{zero} = \mathsf{zero} \Rightarrow \mathsf{zero} = \mathsf{Bplus}(\mathsf{zero}, \mathsf{zero})$$
$$\mathsf{Hb'} : \forall n : \mathsf{bin}.\ n = \mathsf{Bplus}(n, \mathsf{zero}) \Rightarrow \mathsf{S'}(n) = \mathsf{Bplus}(\mathsf{S'}(n), \mathsf{zero})$$

These conjectures are always easily proven automatically, by simply applying rewriting operations repeatedly with the characteristic equations proven in section 6.3. In *Coq* version 6.3.1, the proof is done using the proof command:

```
Repeat (Intro; AutoRewrite [simplification]);Auto.
```

This proof command assumes that users have added all the characteristic equations in the theorem database named simplification as they are proven.

6.5 Bookkeeping

As correspondences are established between functions and theorems in one setting and functions and theorems in the other setting, we need to maintain a lookup table that is used to perform the translation of statements and proofs from one setting to the other. From what we have seen so far, this lookup table has the following shape:

unary setting	*binary setting*	*unary setting*	*binary setting*
nat	bin	nat_ind	new_bin_ind
O	zero	S	S'
plus	Bplus	Ha	Ha'
Hb	Hb'		

This table is augmented every time a conjecture or a theorem is proven. The user may also explicitly add new elements that are provided manually, for instance when implementing multiplication and the characteristic equations for this operation. Here again, efficient multiplication on the binary structure is very different from multiplication on the unary structure.

If all the conjectures have been proven, all the theorems used by a given theorem have already been translated and proven, and the proof term for this theorem does not contain `Case` constructs, then the proof automatically obtained by translation according to the lookup table is always well-typed. Of course there remains the problem of handling the presence of `Cases` constructs. This is the object of the next section.

7 Case Constructs

Case analysis operators can appear in a proof term for various reasons. They are introduced if the proof of a logical property is performed by case analysis on an element of an inductive data type; they appear if proofs involve the one-to-one and disjointness properties of the constructors of an inductive data type; they are also used to describe computation.

We study under which conditions these case analysis steps can be treated and how they are handled in a very pragmatic way. In most cases, structural analysis can be replaced with the application of an induction theorem.

7.1 Translating the "Case" Tactic on a Goal of Sort Prop

In this case, we have a goal, say $P(n)$ to prove and we decide to proceed by case analysis on the element n which belongs to an inductively defined data type. In the example of unary numbers, we replace the `Cases` construct, for instance:

$$\texttt{Cases}\ n\ \texttt{of}\ \mathsf{O} => \mathsf{h0} \mid (\mathsf{S}\ p) => (\mathsf{hr}\ p)\ \texttt{end}$$

by the application of the induction principle

$$(\mathsf{nat\_ind}\ [n : \mathsf{nat}](P\ n)\ h0\ [n : \mathsf{nat}][\_ : (P\ n)](hr\ p))$$

There is no step hypothesis since we perform case analysis rather than induction on the data. The translation of this expression in the binary integers setting is straightforward, replacing nat_ind with new_bin_ind.

7.2 Translating Injectivity and Disjointness Properties of Constructors

The problem with injectivity and disjointness of constructors is that usual proofs for these properties rely on the ι-reduction behavior of the recursors. For instance, the recursor for natural numbers has the following behavior:

$$\begin{aligned}(\mathsf{nat\_rec}\ P\ h0\ hr\ O) &\rightarrow h0\\ (\mathsf{nat\_rec}\ P\ h0\ hr\ (S\ p)) &\rightarrow (hr\ p\ (\mathsf{nat\_rec}\ P\ h0\ hr\ p))\end{aligned}$$

We have not been able to construct an object simulating this behavior in the binary setting. On the other hand, we have been able to recognize injectivity and disjointness proofs by pattern-matching. The patterns for these reasoning steps are respectively:

$$(\mathsf{LET}\ ?\ ?\ ?\ (\mathsf{f\_equal}\ \ldots)\ldots) \quad (\mathsf{LET}\ ?\ ?\ ?\ (\mathsf{eq\_ind}\ \ldots)\ \ldots)$$

Then we build a new proof term by repeatedly applying the two following statements, that are easily proven using their counterparts in the unary setting and the bijections NtoB and BtoN:

$$\begin{aligned}\mathsf{bin\_inj} : \forall n, m : \mathsf{bin}.\quad & (\mathsf{S'}\ n) = (\mathsf{S'}\ m) \rightarrow n = m\\ \mathsf{bin\_discr} : \forall n : \mathsf{bin}.\quad & \neg\mathsf{zero} = (\mathsf{S'}\ n)\end{aligned}$$

7.3 The "Case" Tactic on a Object of Sort Set

This use of the "Case" tactic to build a computational object is an extension of the previous case for injectivity and disjointness. It can be transformed into a `nat_rec` theorem application. However, as we can not provide the user with a suitable translation of such reduction steps, we can not translate such constructs for the time being.

8 Conclusion

We have implemented an ML module that performs the technique presented in this paper automatically. It has been tested on a small collection of theorems (e.g. commutativity and associativity of addition) that have all been translated automatically.

Researchers in the field of programming languages have already considered the issue of concrete data type representation and correctness of abstraction. Among them, Wadler proposed a new concept called *Views* [17] in order to reconcile the conflict between data abstraction and pattern matching in programming languages.

Other researchers have studied the topic of proof transformation. In particular, Madden [10] and Anderson [1] show how automatic transformation of proofs can lead to optimized programs extracted from these proofs. Richardson [15] even uses this technique to change data structures in a proof and the program that is extracted from this proof. His tool automatically obtains an algorithm for addition in the binary setting, but this algorithm is less efficient than the hand-crafted algorithm we study.

Another related topic revolves around proofs by analogy. Informal mathematics contain frequently formulas of the form *One would show P in the same manner that Q*. Melis and Whittle [13] propose a technique to construct a proof of a theorem from a model. Their approach is distinguished by the level of abstraction that they take. They rely on *proof-plans* that basically are very high-level proof procedures.

All this work around proof transformation is obviously related to the problem of transferring proofs from one theorem prover to another. Felty and Howe [7] show that proof transformations make it possible to use results established in different provers at the same time. Denney [6] proposes a mechanism to generate proof scripts for *Coq* from proof descriptions given in HOL.

As our plan was to provide a practical tool usable in the *Coq* system, our study mainly focuses on *Coq* Type Theory features. We did not take into account some other Type Theory features which are not available in the *Coq* system. In particular, we did not try to use the idea of recursion-induction, or tools described in McBride's PhD thesis [12] to derive "induction principles" which are direct elimination rules for function instances.

Our solution that goes through a translation of proof terms is not the only solution available. An alternative is to rely more on the bijection between the two representations, thus generalizing the way we have proven the induction principle new_bin_ind in this paper. If this approach is taken, then one needs to show that the bijections between the two representations establish a morphism with respect to all the operations that take part in the translation. Thus, it is necessary to prove a theorem that has the following statement:

$$\mathsf{morphism\_plus} : \forall x, y : \mathsf{bin}.\ \mathsf{BtoN}(\mathsf{Bplus}(x, y)) = \mathsf{plus}(\mathsf{BtoN}(x), \mathsf{BtoN}(y))$$

Fortunately, this theorem is easy to obtain from the proofs of the characteristic equations as done in section 6.3 and the induction principle new_bin_ind. Once the isomorphisms are established, proving the translated theorems is relatively easy: one simply needs to replace every variable x of type bin in the statement with the term $\mathsf{NtoB}(\mathsf{BtoN}(x))$ and then use the initial theorem by specializing it on $\mathsf{BtoN}(x)$. Complications occur when the statement of the theorem contains elaborate inductive types, conjunctions, disjunctions, existential quantifications. We still have to study this method and compare its applicability with the method described here.

One area where the two methods may have different advantages is the case when the initial setting and the final setting are not exactly isomorphic. For instance, we still need to compare this with an implementation of natural numbers where they are really represented with lists of boolean values. In this case, two different lists may represent the same natural numbers. What happens here is that the syntactic equality, often called Leibnitz's equality, is in correspondence with an arbitrary equivalence relation: terms that can be distinguished in the final setting correspond to identical terms in the initial setting.

References

1. Penny Anderson. Representing proof transformations for program optimization. In *International Conference on Automated Deduction*, LNAI 814, Springer-Verlag, 1994.
2. Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In Harrison and Aagaard [9], pages 1–16.
3. B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997. revised version distributed with Coq.
4. S. Boutin. Using reflection to build efficient and certified decision procedures. In Martin Abadi and Takahashi Ito, editors, *TACS'97*, LNCS 1281, Springer-Verlag, 1997.
5. Yann Coscoy. A natural language explanation for formal proofs. In Christian Rétoré, editor, *Logical Aspects of Computational Linguistics*, LNAI 1328, Springer-Verlag, 1996.
6. Ewen Denney. A prototype proof translator from HOL to Coq. In Harrison and Aagaard [9], pages 108–125.
7. Amy P. Felty and Douglas J. Howe. Hybrid interactive theorem proving using nuprl and hol. In *International Conference on Automated Deduction*, number 1249 in LNAI. Springer-Verlag, 1997.
8. H. Geuvers, F. Wiedijk, and J. Zwanenburg. Equational reasoning via partial reflection. In Harrison and Aagaard [9], pages 163–179.
9. J. Harrison and M. Aagaard, editors. *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, LNCS 1869, Springer-Verlag, 2000.
10. Peter Madden. The specialization and transformation of constructive existence proofs. In N.S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 131–148. Morgan Kaufmann, 1989.
11. Lena Magnusson. *The Implementation of ALF - a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitutions.* PhD thesis, Chalmers University of Technology / Göteborg University, 1995.

12. Conor McBride. *Dependently Typed Functional Programs and their Proofs.* PhD thesis, University of Edinburgh, 1999.
13. Erica Melis and Jon Whittle. Analogy in inductive theorem proving. *Journal of Automated Reasoning*, 22:117–147, 1999.
14. Christine Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, LNCS 664, Springer-Verlag, 1993. LIP research report 92-49.
15. Julian Richardson. Automating changes of data type in functional programs. In *Proceedings of KBSE-95.* IEEE Computer Society, 1995.
16. Simon Thompson. *Type Theory and functional Programming.* Addison-Wesley, 1991.
17. Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings, 14th Symposium on Principles of Programming Languages POPL'87.* ACM, 1987.