Changing Data Representation within the Coq System

Nicolas Magaud

INRIA Sophia-Antipolis, France

Abstract. In a theorem prover like Coq, mathematical concepts can be implemented in several ways. Their different representations can be either efficient for computing or well-suited to carry out proofs easily. In this paper, we present improved techniques to deal with changes of data representation within Coq. We propose a smart handling of case analysis and definitions together with some general methods to transfer recursion operators and their reduction rules from one setting to another. Once we have built a formal correspondence between two settings, we can translate automatically properties obtained in the initial setting into new properties in the target setting. We successfully experiment with changing Peano's numbers into binary numbers for the whole *Arith* library of Coq as well as with changing polymorphic lists into reversed (snoc) lists.

1 Introduction

In this paper, we present a general method to change the way we look at a data type in a proof system such as Coq [5], and to enhance proof reuse when shifting from a data type representation to another [17].

For instance, one may want to switch from Peano's encoding of natural numbers to a binary representation. This change may be motivated by efficiency reasons, computations are much faster with binary integers than with unary integers. We provide a smart mechanism to avoid proving again in the final setting properties already established in the initial setting.

Previously in [12], we presented a first experiment about changes of data type representation in type theory. This work was limited to equational reasoning. We did not propose any general way to handle structural case analysis and therefore inversion techniques [6] properly. In addition, we did not consider inductive predicates and definition unfolding for logical properties. The present work aims at removing these weaknesses. We choose to develop a practical tool usable in the Coq system. As a consequence, our experiments were restricted to the Calculus of Inductive Constructions [15] and its implementation in the Coq system. Therefore we cannot easily use techniques such as induction-recursion [9] or those proposed in [13] for instance.

Throughout this paper, we will use the example of natural numbers and consider their unary and binary representations. However, we would like to emphasize that we also experimented with changing the data representation for polymorphic lists.

D. Basin and B. Wolff (Eds.): TPHOLs 2003, LNCS 2758, pp. 87–102, 2003.

1.1 Related Work

Translating proofs from one setting to another require abstracting away most of the implementation of a mathematical concept. Viewing concrete datatypes in a more abstract way can be achieved in many different ways.

- The Coq system provides some tools to deal with proofs in some algebraic settings, for instance ring [5, Chap. 19] and field [7] structures.
- Modules and functors [4], as those introduced in the latest version of Coq make it possible to have a high-level view of mathematical concepts we manipulate. It allows the user to hide the actual implementation of a data type. Signatures (module types) can be easily instantiated with various concrete representations.
- G. Barthe and O. Pons [2] suggest to use type isomorphisms to enhance proof reuse in dependent type theory. They give a computational interpretation of some type isomorphisms using coercions. Their work allows the user to view mathematical objects from various points of view.

1.2 An Example

We consider the statement $\forall n : \mathsf{nat}.\ n \leq O \rightarrow n = O$ as an example. It is interesting because it was not processed properly by the method proposed in [12]. It can be proved in Coq by running the following script:

```
Lemma example : (n:nat)(le n O)->n=O.
Intros n H ; Inversion H ; Auto.
Qed.
```

The proof term generated from this script is shown in figure 1. The tactic *Inversion H* builds a term which features case analysis on an instance of an inductive data type $H : (\mathsf{le}\ n\ \mathsf{O})$. In addition, it contains structural case analysis on nat, intended to discriminate the assumption $H_1 : ((\mathsf{S}\ m) = \mathsf{O})$. We propose new methods and tools to transfer this proof from the unary setting into the binary one. We mainly focus on handling structural case analysis in a smart manner.

1.3 Outline

In section 2, we present definitions of inductive datatypes and their associated recursion operators within Coq. In section 3, we introduce the language we consider for proof terms. In section 4, we show how to remove case constructs for proof terms. In section 5, we study the issue of definition unfolding, especially for types containing logical information. In section 6, we present the implementation of our tool and two case studies. Finally, in section 7, we give some perspectives about related and future work.

```
λn : nat.λH : (le n O).
  let H0 = ⟨λp : nat.p = O → n = O⟩
           Cases H of
              le_n ⇒ λH0 : n = O.
                 (eq_ind nat O λn : nat.n = O (refl_equal nat O)
                         n (sym_eq nat n O H0))
            | (le_S m H0) ⇒ λH1 : (S m) = O.
                 (let H2 = (eq_ind nat (S m)
                              λe : nat.⟨Prop⟩Cases e of
                                             O ⇒ False
                                           | (S _) ⇒ True
                                           end I O H1)
                 in (False_ind (le n m) → n = O H2) H0)
           end
  in (H0 (refl_equal nat O)).
```

Fig. 1. A term proving the lemma example

2 Context of This Work

In this section, we first present the way recursion operators are defined in the Coq system. Then we recall how we proceed to make conversions explicit in a proof term [12].

2.1 Recursion Operators in the Calculus of Constructions

From an inductive definition, e.g.

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

the Coq system automatically generates structural recursion operators (one for each sort Set, Type, Prop). For instance nat_rec has the following statement:

$$\mathsf{nat\_rec} : \forall P : \mathsf{nat} \to \mathsf{Set}.\ (P\ O) \to (\forall n : \mathsf{nat}.\ (P\ n) \to (P\ (S\ n))) \to \forall n : \mathsf{nat}.\ (P\ n)$$

On the other hand, the definition of an inductive predicate such as le only triggers the construction of a single recursion operator le_ind.

```
Inductive le [n : nat]  : nat->Prop :=
      le_n : (le n n) | le_S : (m:nat)(le n m)->(le n (S m))
```

$$\begin{aligned}\mathsf{le\_ind} : {} & \forall n : \mathsf{nat}.\ \forall P : \mathsf{nat} \to \mathsf{Prop}.\\ & (P\ n) \to (\forall m : \mathsf{nat}.\ (\mathsf{le}\ n\ m) \to (P\ m) \to (P\ (S\ m))) \to\\ & \forall n0 : \mathsf{nat}.\ (\mathsf{le}\ n\ n0) \to (P\ n0)\end{aligned}$$

Indeed, elimination rules do not allow building elements of sorts Type or Set by case analysis on an element of sort Prop.

Dependent vs. Non-dependent Recursion Operators. The Coq system allows the definition of two kinds of induction principles: dependent (also called maximal) and non-dependent (minimal) ones [3].

As an example, in addition to the dependent recursion operator for nat, we give the non-dependent one. It corresponds to the recursion operator of Gödel's system T. nat_min_rec states that:

$$\forall P : \mathsf{Set}.\ P \to (\mathsf{nat} \to P \to P) \to \mathsf{nat} \to P \tag{1}$$

Such a recursor can be used to define basic operations such as plus easily:

$$\begin{aligned}\mathsf{plus'} \ = \ & \lambda n : \mathsf{nat}.(\mathsf{nat\_min\_rec}\ \mathsf{nat} \to \mathsf{nat} \\ & \lambda m : \mathsf{nat}.m \\ & \lambda\_ : \mathsf{nat}.\lambda vr : \mathsf{nat} \to \mathsf{nat}.\lambda m : \mathsf{nat}.(\mathsf{S}\ (vr\ m)) \\ & n)\end{aligned}$$

In figure 2, we give the characteristics of the recursion operators automatically inferred after the definition of an inductive type T. The Coq proof assistant defines different recursion operators to build objects of sorts Set, Type, Prop. Columns of this array represent the sort of the element which is applied to the recursion operator and rows the sort of the element it builds.

	T : Set or T : Type	T : Prop
$P : T \to$ Set or $P : T \to$ Type	dependent	not allowed
$P : T \to$ Prop	dependent	non dependent

Fig. 2. Allowed eliminations

To sum up, the definition of nat whose sort is Set triggers the automatic definition of three dependent elimination principles nat_rec, nat_ind and nat_rect. On the other hand, the definition of le triggers the definition of a single elimination principle le_ind which is a non dependent one.
Anyway, in both cases, non-dependent (resp. dependent) principles can also be defined in addition to their dependent (resp. non-dependent) counterparts using the Scheme command:

$$\mathsf{Scheme}\ T_{(\mathsf{ind}|\mathsf{rec}|\mathsf{rect})} := (\mathsf{Induction} \mid \mathsf{Minimality})\ \mathsf{for}\ T\ \mathsf{Sort}\ (\mathsf{Prop} \mid \mathsf{Set} \mid \mathsf{Type}).$$

For instance, the non-dependent induction principle (1) for nat can be generated by the command: `Scheme nat_min_rec := Minimality for nat Sort Set.`

Recursion Operators and Their Reduction Rules. All these recursion operators are defined using fix-point and case analysis constructs. Dependent and non-dependent operators are defined in the same manner. For instance nat_rec is defined as follows:

```
nat_rec = [P:(nat->Set); f:(P (0)); f0:((n:nat)(P n)->(P (S n)))]
 Fix F {F [n:nat] : (P n) :=
          <P>Cases n of (0) => f
                      | (S n0) => (f0 n0 (F n0))
             end}
```

As a consequence of their definitions, recursion operators have a computational behavior. For nat_rec, it can be expressed by these two reduction rules:

$$\mathsf{nat\_rec}\ P\ v0\ vr\ O \xrightarrow{\iota} v0$$
$$\mathsf{nat\_rec}\ P\ v0\ vr\ (S\ n) \xrightarrow{\iota} vr\ n\ (\mathsf{nat\_rec}\ P\ v0\ vr\ n)$$

2.2 Convertibility Issues

In this section, we sum up the main results of the work presented in [12]. The aim of this work was to exhibit implicit proof steps inside proof terms. The algorithm we proposed at the time did not provide any support for handling structural case analysis.

Making Conversions Explicit. We consider the theorem plus_n_O and show how it is processed by the algorithm (see [12] for details). This theorem states that:

$$\forall n \in \mathsf{nat}.\quad n = (\mathsf{plus}\ n\ \mathsf{O}) \tag{2}$$

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

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

It proceeds by induction on n, using the principle nat_ind. The base case requires proving that $\mathsf{O} = (\mathsf{plus}\ \mathsf{O}\ \mathsf{O})$. The step case requires proving that

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

The term (refl_equal nat O) is a proof of the base case. However, the inferred (or proposed) type for this term is $\mathsf{O} = \mathsf{O}$, whereas its expected type is $\mathsf{O} = (\mathsf{plus}\ \mathsf{O}\ \mathsf{O})$. These two types are convertible thanks to the computational rules derived from the definition of plus, but they are not syntactically equal.

In our example, steps which are made explicit by the algorithm are formalized by the conjectures Ha and Hb whose statements are:

$$Ha : \mathsf{O} = \mathsf{O} \Rightarrow \mathsf{O} = (\mathsf{plus}\ \mathsf{O}\ \mathsf{O})$$
$$Hb : \forall n : \mathsf{nat}.\ (\mathsf{S}\ n) = (\mathsf{S}\ (\mathsf{plus}\ n\ \mathsf{O})) \Rightarrow (\mathsf{S}\ n) = (\mathsf{plus}\ (\mathsf{S}\ n)\ \mathsf{O})$$

These conjectures aim at connecting expected and proposed types in the branches of the induction principle. 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. Eventually, the algorithm returns the following proof term:

$$\begin{aligned}
&\lambda n : \mathsf{nat}.(\mathsf{nat_ind}\\
&\quad (\lambda n0 : \mathsf{nat}.n0 = (\mathsf{plus}\ n0\ \mathsf{O}))\\
&\quad (Ha\ (\mathsf{refl_equal\ nat\ O}))\\
&\quad \lambda n0 : \mathsf{nat}; H : (n0 = (\mathsf{plus}\ n0\ \mathsf{O})).\\
&\qquad (Hb\ n0\ (\mathsf{f_equal\ nat\ nat\ S}\ n0\ (\mathsf{plus}\ n0\ \mathsf{O})\ H))\ n)
\end{aligned}$$

Representing Functions. In the target setting of our proof transformation process, addition may have different reduction rules compared to the ones it has in the Peano's setting. As a consequence, we have to make these computations explicit. We express them as equations. Let us consider the example of plus.

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

plus has the following reduction rules:

$$\mathsf{plus\ O}\ m \xrightarrow{\iota} m \qquad\qquad \mathsf{plus\ (S}\ p)\ m \xrightarrow{\iota} \mathsf{S\ (plus}\ p\ m)$$

They are formalized as equations that would be translated and proved in the new setting:

$$\begin{aligned}
\forall m : \mathsf{nat}.\ (\mathsf{plus\ O}\ m) &= m\\
\forall p, m : \mathsf{nat}.\ (\mathsf{plus\ (S}\ p)\ m) &= (\mathsf{S\ (plus}\ p\ m))
\end{aligned}$$

Conjectures Ha and Hb can be proved by rewriting using these two equations. We postpone the actual translation of a proof term into the binary setting until section 4.4. In this section, we have shown how to proceed when terms do not contain case analysis constructs. In the forthcoming sections, we study how to transform a term with case constructs into a term without any. After defining formally the terms we consider, we will show how to proceed in section 4.

3 Terms

As usual in type theory, we consider a set of sorts $\mathcal{S}$ which contains Prop, Set and Type. A *term* is an element of the language $\mathcal{T}$ defined as follows:

$$\begin{aligned}
\mathcal{T} ::= &\ \lambda x : \mathcal{T}.\mathcal{T} \mid (\mathcal{T}\ \mathcal{T}) \mid \forall x : \mathcal{T}.\ \mathcal{T} \mid \mathcal{S} \mid x \mid \mathcal{C} \mid \mathcal{I} \mid \mathcal{CI}\\
&\mid \mathsf{let}\ x\ = \mathcal{T} : \mathcal{T}\ \mathsf{in}\ \mathcal{T}\\
&\mid \langle \mathcal{T} \rangle \mathsf{case}\ \mathcal{T}\ \mathsf{of}\ \{\mathcal{T}\}
\end{aligned}$$

$\mathcal{C}$ denotes the set of constants, $\mathcal{I}$ the set of inductive datatypes and $\mathcal{CI}$ the set of constructors for these inductive datatypes. This corresponds to the usual set of terms that occur in proof terms. We treat recursion as it usually appears in proof terms, i.e. hidden in a constant definition such as nat_ind. Therefore we assume that fix-point constructs do not occur explicitly in proof terms.

The process to transform a proof term is divided into two steps. We first remove all occurrences of case (see section 4); we then extract the conversion steps from the generated proof term by using the approach presented in section 2.2.

4 Removing Structural Case Analysis from Proof Terms

Case expressions can occur in proof terms because the proof script explicitly contains an occurrence of the Case tactic or because an inversion was performed. Inversion gives rise to case analysis whose result type has sort Set or Type. In this setting, reduction rules are necessary, thus it is relevant to translate them.

Structural case analysis is replaced by using a *defined* case analysis operator and its associated reduction rules (if needed). As stated previously for recursion operators, we emphasize that case analysis from Prop to Set or Type is not allowed. The only exception is inductive datatypes with a single constructor e.g. Leibnitz's equality eq.

We assume we have the following inductive definition:

$$\begin{array}{rl} \mathsf{Inductive}\ T\ :\ \mathsf{S}\ := & \mid c1 : \forall t_{1,1} : T_{1,1} \ldots\ \forall t_{1,i_1} : T_{1,i_1}.\ T \\ & \mid \ldots \\ & \mid c_k : \forall t_{k,1} : T_{k,1} \ldots\ \forall t_{k,i_k} : T_{k,i_k}.\ T \\ & \mid \ldots \\ & \mid c_n : \forall t_{n,1} : T_{n,1} \ldots\ \forall t_{n,i_n} : T_{n,i_n}.\ T. \end{array}$$

4.1 Algorithm

We consider a proof p of an arbitrary property on the elements of T. The algorithm performs a recursive structural analysis of the term p. For most of the terms, it merely calls itself recursively on their sub-terms if they have any; otherwise it simply returns the term. The only interesting part is how case analysis is handled. In figure 3, we give an example of a case expression on t of type T. The left-hand side corresponds to the actual notation for case analysis whereas the right-hand side notation is closer to the actual implementation of case analysis in Coq. In presence of a case expression, the algorithm acts as follows:

1. It computes the type of t. Let us assume $t : T$. As seen in section 2.1, we have at most six recursion operators available for T: three dependent ones, namely T_rec, T_ind and T_rect and three non-dependent ones T_min_rec, T_min_ind and T_min_rect.

$$
\begin{array}{l}
\langle P\rangle\ \mathsf{Cases}\ t\ \mathsf{of} \\
\quad\ \ |\ (c_1\ t_{1,1} \ldots\ t_{1,i_1}) \Rightarrow r_1 \\
\quad\ \ |\ldots \\
\quad\ \ |\ (c_k\ t_{k,1} \ldots\ t_{k,i_k}) \Rightarrow r_k \\
\quad\ \ |\ldots \\
\quad\ \ |\ (c_n\ t_{n,1} \ldots\ t_{n,i_n}) \Rightarrow r_n \\
\mathsf{end}
\end{array}
\quad\equiv\quad
\begin{array}{l}
\langle P\rangle\ \mathsf{Case}\ t\ \mathsf{of} \\
\quad \lambda t_{1,1} : T_{1,1} \ldots\ \lambda t_{1,i_1} : T_{1,i_1}.r_1 \\
\quad \ldots \\
\quad \lambda t_{k,1} : T_{k,1} \ldots\ \lambda t_{k,i_k} : T_{k,i_k}.r_k \\
\quad \ldots \\
\quad \lambda t_{n,1} : T_{n,1} \ldots\ \lambda t_{n,i_n} : T_{n,i_n}.r_n \\
\mathsf{end}
\end{array}
$$

Fig. 3. A case expression to be processed by our algorithm

2. As shown in figure 3, P is the elimination predicate associated to a case construct [5, Chap. 14]. Types of the branches of the case construct are instances of this elimination predicate. The shape of P allows us to determine whether we face dependent case analysis or not. If P is a λ-abstraction, we face a dependent case. If not, we face a non-dependent case. It remains to determine the sort of the objects we build. To do so, we compute the type of the value returned by P. For instance, if $P \equiv\ \lambda n : \mathsf{nat}.(\mathsf{le}\ \mathsf{O}\ n) \rightarrow n = \mathsf{O}$ then the type of the returned value is Prop.
3. Once we have the shape and the name of the right recursion operator, (let us call it induction_for_T), we build an instance of the corresponding case analysis operator, say case_for_T. For example, nat_rec is transformed into a (non-recursive) case analysis operator nat_case_rec whose type is:

$$\forall P : \mathsf{nat} \rightarrow \mathsf{Set}.\ (P\ \mathsf{O}) \rightarrow (\forall n : \mathsf{nat}.\ (P\ (\mathsf{S}\ n))) \rightarrow \forall n : \mathsf{nat}.\ (P\ n).$$

4. The algorithm recursively computes new terms $r'_1, \ldots, r'_n$ (with no more structural case expressions) for all sub-terms $r_1, \ldots, r_n$ of the case expression.
5. It remains to build an application whose head is case_for_T and that mimics structural case analysis via application of the *defined* case analysis operator. Eventually, the algorithm returns the following term instead of the case expression.

$$
\begin{array}{l}
(\mathsf{case\_for\_T}\ P\ \ \lambda t_{1,1} : T_{1,1} \ldots\ \lambda t_{1,i_1} : T_{1,i_1}.r'_1 \\
\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad \lambda t_{n,1} : T_{n,1} \ldots\ \lambda t_{n,i_n} : T_{n,i_n}.r'_n \\
\qquad\qquad\qquad t)
\end{array}
$$

This transformation removes explicit structural case analysis and hides it in the application of a defined case analysis operator. It provides us with a way to reason by case analysis without knowing anything about the actual inductive representation of the data type.

4.2 What Happens to Our Example ?

If we consider the example shown in figure 1, *Cases H of ...* has been replaced by the application of the case analysis principle le_case_ind. In addition,

Cases e of ... has been replaced by the application of the non-dependent operator nat_case_rect_min whose type is $\forall P : \mathsf{Type}.\ P \to (\mathsf{nat} \to P) \to \mathsf{nat} \to P$.

$\lambda n : \mathsf{nat}.\lambda H : (\mathsf{le}\ n\ \mathsf{O}).$
  $let\ H_0\ =(\mathsf{le\_case\_ind}\ n\ \lambda p : \mathsf{nat}.p = \mathsf{O} \to n = \mathsf{O}$
    $\lambda H_0 : n = \mathsf{O}.$
      $(\mathsf{eq\_ind}\ \mathsf{nat}\ \mathsf{O}\ \lambda n : \mathsf{nat}.n = \mathsf{O}\ (\mathsf{refl\_equal}\ \mathsf{nat}\ \mathsf{O})$
        $n\ (\mathsf{sym\_eq}\ \mathsf{nat}\ n\ \mathsf{O}\ H_0))$
    $\lambda m : \mathsf{nat}.\lambda H_0 : (\mathsf{le}\ n\ m).\lambda H_1 : (\mathsf{S}\ m) = \mathsf{O}.$
      $(let\ H_2\ =\ (\mathsf{eq\_ind}\ \mathsf{nat}\ (\mathsf{S}\ m)$
        $\lambda e : \mathsf{nat}.$
        $(\mathsf{nat\_case\_rect\_min}\ \mathsf{Prop}\ \mathsf{False}\ \lambda\_ : \mathsf{nat}.\mathsf{True}\ e)$
        $\mathsf{I}\ \mathsf{O}\ H_1)$
       $in\ (\mathsf{False\_ind}\ (\mathsf{le}\ n\ m) \to n = \mathsf{O}\ H_2)\ H_0)$
    $\mathsf{O}\ \mathsf{H})$
  $in\ (H_0\ (\mathsf{refl\_equal}\ \mathsf{nat}\ \mathsf{O})).$

4.3 Actual Translation into a New Representation

Once we have replaced a case expression by the application of a function, we need to translate this function into the target setting. In this section, we present a binary representation of natural numbers and show how to transfer recursion operators from the Peano setting into the binary one.

Binary Representation of Natural Numbers

```
Inductive pos : Set := one: pos          (* 1 *)
                     | pI :pos -> pos    (* 2x+1, x>0 *)
                     | p0: pos -> pos.   (* 2x, x>0 *)
Inductive bin : Set := b0: bin | bp: pos -> bin.
```

From these inductive types, we define counterparts of O and S, namely b0 and bS, as well as translation functions from nat to bin (n2b) and vice-versa (b2n). These definitions are taken from [12, pp. 189-190].

Recursion Operators in the New Setting. We can prove non-dependent recursion operators and their associated reduction rules as equations via the isomorphism connecting nat and bin. We use the equation $\forall n : \mathsf{bin}.\ (\mathsf{n2b}\ (\mathsf{b2n}\ n)) = n$ and the recursion principle for nat nat_rec_min to define the recursion operator $\mathsf{new\_bin\_rec\_min} : \forall P : \mathsf{Set}.\ P \to (\mathsf{bin} \to P \to P) \to \mathsf{bin} \to P$ and prove its properties:

$$(\mathsf{new\_bin\_rec\_min}\ T\ v0\ vr\ \mathsf{b0}) = v0$$
$$\forall p : \mathsf{bin}.\ (\mathsf{new\_bin\_rec\_min}\ T\ v0\ vr\ (\mathsf{bS}\ p)) = (vr\ p\ (\mathsf{new\_bin\_rec\_min}\ T\ v0\ vr\ p))$$

However, as far as dependent recursion principles are concerned, rewriting gets stuck when trying to prove the reduction rules as equations. Therefore we choose to define the dependent recursion operators by well founded induction over binary integers. We establish their reduction rules as a fix-point equation following the technique proposed by A. Balaa and Y. Bertot in [1].

Building the Dependent Recursion Operator

1. First of all, we derive the order on binary integers from the one on Peano's numbers: (Lt x y) $\equiv$ (lt (b2n x) (b2n y)). We show using the standard library of Coq that Lt is well-founded. From now on, this theorem will be named wf_Lt.
2. We then show that a binary integer p is either b0 or (bS q) for some q. This property can be stated with the following dependent inductive definition:

```
Inductive Pred_spec: bin ->Set :=
      is_zero: (Pred_spec b0)
    | is_S: (y:bin)(Pred_spec (bS y)).
```

3. The next step consists in defining a function Pred. Given a binary integer n, it computes an element of type (Pred_spec n) which contains the predecessor of n, when n is not equal to b0. Pred is defined using a proof mode style. It makes it easier to treat dependent case analysis.

```
Definition Pred: (x:bin)(Pred_spec x).
```

4. Once the Pred function is defined, we build a higher-order function F. It takes as input a binary integer n and a function $f : \forall m \in \mathsf{bin}.\ m < n \rightarrow (P\ m)$ and computes an element of type $(P\ n)$. This function performs structural case analysis on (Pred n) and uses the hypotheses $\mathsf{h_0} : (P\ \mathsf{b0})$ and $\mathsf{h_r} : \forall n \in \mathsf{bin}.\ (P\ n) \rightarrow (P\ (\mathsf{bS}\ n))$ which correspond to the premises of the recursion operator for Peano's numbers.

```
Definition F: (n:bin) ((m:bin)(Lt m n)->(P m)) ->(P n) :=
 [n:bin]
  <[n:bin] [p:(Pred_spec n)] ((m:bin)(Lt m n)->(P m)) ->(P n)>
    Cases (Pred n) of
      is_zero => [f:(m:bin)(Lt m b0)->(P m)]h0
    | (is_S p) =>
      [f:(m:bin)(Lt m (bS p))->(P m)](hr p (f p (S_and_Lt p)))
    end.
```

S_and_Lt is a proof that $\forall p : \mathsf{bin}.\ (\mathsf{Lt}\ p\ (\mathsf{bS}\ p))$.

5. Finally, we get our expected recursion operator by the following definition:

```
Definition new_bin_rec :=
               (well_founded_induction bin Lt wf_Lt P F).
```

unary setting	*binary setting*	*unary setting*	*binary setting*
nat	bin	O	b0
plus	bplus	S	bS
nat_ind	new_bin_ind	nat_case_ind	new_bin_case_ind
nat_rect_min	new_bin_rect_min	nat_case_rect_min	new_bin_case_rect_min
le	ble	lt	blt
le_ind	ble_ind	le_case_ind	ble_case_ind

Fig. 4. Bookkeeping

Reduction Rules for the Dependent Recursion Operator. Recursion operators should be considered as common functions. As for plus in section 2.2, we have to state and prove their reduction rules as equations. To do so, we build an instance of the so-called *step hypothesis* of the *transfer theorem* as presented in [1, page 5].

$\forall x : \mathsf{bin}.$
$\forall f' : \forall y : \mathsf{bin}.\ (P\ y).$
$\forall g : \forall y : \mathsf{bin}.\ (\mathsf{Lt}\ y\ x) \to (P\ y).$
$(\forall y : \mathsf{bin}.\ \forall h : (\mathsf{Lt}\ y\ x).\ (g\ y\ h) =_{(P\ y)} (f'\ y)) \to$
$(\mathsf{F}\ x\ \lambda y : bin.\lambda h : (\mathsf{Lt}\ y\ x).(g\ y\ h)) =_{(P\ x)} (\mathsf{F}\ x\ \lambda y : \mathsf{bin}.\lambda\_ : (\mathsf{Lt}\ y\ x).(f'\ y)).$

The proof is carried out by case analysis on (Pred x). The application of the *transfer theorem* leads to the general fix-point equation step_rec:

$$\forall n : \mathsf{bin}\ .(\mathsf{new\_bin\_rec}\ n) = (\mathsf{F}\ n\ \lambda m : \mathsf{bin}.\lambda h : (\mathsf{Lt}\ m\ n).(\mathsf{new\_bin\_rec}\ m))$$

To get the simplified form of the fix-point equation, we need to prove the following lemmas:

$$\forall v : (\mathsf{Pred\_spec}\ b0).\ v = \mathsf{is\_zero}$$
$$\forall n : \mathsf{bin}.\ \forall v : (\mathsf{Pred\_spec}\ (\mathsf{bS}\ n)).\ v = (\mathsf{is\_S}\ n)$$

The first one is trivial whereas the second one requires the use of a dependent equality and dependent inversion techniques. Once P, h_0 and h_r have been discharged, the resulting equations are:

$(\mathsf{new\_bin\_rec}\ P\ h_0\ h_r\ b0) = h_0$
$\forall n : \mathsf{bin}.\ (\mathsf{new\_bin\_rec}\ P\ h_0\ h_r\ (\mathsf{bS}\ n)) = (h_r\ n\ (\mathsf{new\_bin\_rec}\ P\ h_0\ h_r\ n))$

4.4 Back to Our Example

After extracting conversion steps (see section 2.2), and syntactically replacing all objects in the unary setting by their counterparts in the binary one (see figure 4), we get the following proof term:

$$
\begin{array}{l}
\lambda n : \mathsf{bin}.\lambda H : (\mathsf{ble}\ n\ \mathsf{b0}).\\
\quad \mathit{let}\ H_0\ = (\mathsf{ble\_case\_ind}\ n\ \lambda p : \mathsf{bin}.p = \mathsf{b0} \to n = \mathsf{b0}\\
\qquad\qquad \lambda H_0 : n = \mathsf{b0}.\\
\qquad\qquad\quad (\mathsf{eq\_ind}\ \mathsf{bin}\ \mathsf{b0}\ \lambda n : \mathsf{bin}.n = \mathsf{b0}\ (\mathsf{refl\_equal}\ \mathsf{bin}\ \mathsf{b0})\\
\qquad\qquad\qquad\qquad n\ (\mathsf{sym\_eq}\ \mathsf{bin}\ n\ \mathsf{b0}\ H_0))\\
\qquad\qquad \lambda m : \mathsf{bin}.\lambda H_0 : (\mathsf{ble}\ n\ m).\lambda H_1 : (\mathsf{S}\ m) = \mathsf{b0}.\\
\qquad\qquad\quad (\mathit{let}\ H_2\ = \ (\mathbf{example\_rr2}\ (\mathsf{eq\_ind}\ \mathsf{bin}\ (\mathsf{bS}\ m)\\
\qquad\qquad\qquad\qquad \lambda e : \mathsf{bin}.\\
\qquad\qquad\qquad\qquad (\mathsf{new\_bin\_case\_rect\_min}\ \mathsf{Prop}\ \mathsf{False}\ \lambda\_ : \mathsf{bin.True}\ e)\\
\qquad\qquad\qquad\qquad (\mathbf{example\_rr1}\ m\ \mathsf{I})\ \mathsf{b0}\ H_1))\\
\qquad\qquad\quad \mathit{in}\ (\mathsf{False\_ind}\ (\mathsf{ble}\ n\ m) \to n = \mathsf{b0}\ H_2)\ H_0)\\
\qquad\qquad \mathsf{b0}\ \mathsf{H})\\
\quad \mathit{in}\ (H_0\ (\mathsf{refl\_equal}\ \mathsf{bin}\ \mathsf{b0})).
\end{array}
$$

Its type is $\forall n : \mathsf{bin}.\ (\mathsf{ble}\ n\ \mathsf{b0}) \to n = \mathsf{b0}$. The conjectures example_rr1 and example_rr2 are generated automatically to connect expected and proposed terms for sub-expressions of this proof term. The term example_rr1 is a proof of

$$\forall m : \mathsf{bin}.\ \mathsf{True} \to (\mathsf{new\_bin\_case\_rect\_min}\ \mathsf{Prop}\ \mathsf{False}\ \lambda\_ : \mathsf{bin.True}\ (\mathsf{bS}\ m)) \quad (3)$$

and example_rr2 a proof of

$$(\mathsf{new\_bin\_case\_rect\_min}\ \mathsf{Prop}\ \mathsf{False}\ \lambda\_ : \mathsf{bin.True}\ \mathsf{b0}) \to \mathsf{False} \quad (4)$$

The statements (3) and (4) are proven by rewriting with the fix-point equations for new_bin_case_rect_min corresponding to reduction rules for nat_case_rect_min in the binary setting.

5 Constants Unfolding and Changes of Representations

5.1 A Basic Example

Operations in the binary setting are designed to be as efficient as possible. Therefore definitions of operations on binary integers are not straightforward translations of the ones in Peano's arithmetics.

In order to be able to reuse the proofs, we need to connect the new definition to the previous one. As an example, we consider the function which computes the double of a number.

$$\mathsf{double}\ =\ \lambda n : \mathsf{nat}.(\mathsf{plus}\ n\ n)$$

Let us suppose we already proved and want to translate the following theorem:

$$\forall n, m : \mathsf{nat}.\ (\mathsf{double}\ (\mathsf{plus}\ n\ m)) = (\mathsf{plus}\ (\mathsf{double}\ n)\ (\mathsf{double}\ m)) \quad (5)$$

In the binary setting bdouble is defined as follows:

$$\mathsf{bdouble} = \lambda n : \mathsf{bin}.\mathit{Cases}\ n\ \mathit{of}\ \mathsf{b0} \Rightarrow b0 \mid (\mathsf{bp}\ p) \Rightarrow (\mathsf{bp}\ (\mathsf{pO}\ p))\ \mathit{end}$$

A proof of (5) may rely on the property that (double n) and (plus n n) are δ-convertible. This convertibility rule can be mimicked by the following equation:

$$\forall n : \mathsf{bin}.\ (\mathsf{bdouble}\ n) = (\mathsf{bplus}\ n\ n)$$

Such an equation will be useful to prove by rewriting the conjectures generated when making conversions explicit (see section 2.2).

5.2 A More Tedious Example

We now consider the definition of the relation lt:

$$\mathsf{lt} = \lambda n, m : \mathsf{nat}.(\mathsf{le}\ (\mathsf{S}\ n)\ m)$$

As a consequence of the definitions of le and lt, (le (S n) m) and (lt n m) are convertible whereas there is no reason for lt to be defined in terms of le. This becomes an issue because blt will not necessarily be defined in terms of ble in the binary setting, therefore this implicit equality will be lost. Our first idea was to proceed in the same way as above. But a problem arises. In the binary setting, we can not prove

$$\forall n, m : \mathsf{bin}.\ (\mathsf{ble}\ (\mathsf{bS}\ n)\ m) == (\mathsf{blt}\ n\ m) \tag{6}$$

where == denotes Leibnitz's equality in Type [1]. Trying to prove this statement, we got into a red herring. Because the definitions of blt and ble use Leibnitz's equality, we have to prove something like (true=true)==(¬false=true). This happens to be unprovable in the Coq system.
However we can easily establish the equivalence property:

$$\forall n, m : \mathsf{bin}.\ (\mathsf{ble}\ (\mathsf{bS}\ n)\ m) \leftrightarrow (\mathsf{blt}\ n\ m) \tag{7}$$

where $\leftrightarrow$ is defined as $A \leftrightarrow B \equiv (A \to B) \wedge (B \to A)$. Unfortunately, one can not use this property easily. Indeed, we want to be able to substitute (ble (bS n) m) and (blt n m) and one can not directly rewrite using the equivalence relation.
In the rest of this section, we consider two options to handle this question. First, we assume the axiom $\forall P, Q : \mathsf{Prop}.\ (P \leftrightarrow Q) \to P == Q$ is valid in the calculus of constructions. Therefore the first equation (6) holds and we can perform rewriting easily. The second option is to consider the Setoid (Prop,$\leftrightarrow$) and perform setoid-rewriting.

5.3 Handling Propositional Equivalence as Leibnitz's Equality

It is safe [14] to add the following axiom in the Coq system.

$$\mathsf{Axiom\ iff\_implies\_equiv}\ : \forall P, Q : \mathsf{Prop}.\ (P \leftrightarrow Q) \to P == Q$$

However, in this work, we always use a definitional approach without any parameter or axiom. In addition, all translated proofs are checked by the Coq kernel before being accepted. Therefore we preferred not to use an axiom and we used Setoid rewriting with (Prop,$\leftrightarrow$), thus keeping a definitional approach.

[1] `Inductive eqT [A : Type; x : A] : A->Prop := refl_eqT : x==x.`

5.4 Rewriting with Setoids

Coq provides a tactic Setoid_rewrite [5, Chap. 20] which is intended to perform rewriting with an equivalence relation. We consider the setoid (Prop,$\leftrightarrow$) and experiment how to use it to substitute (blt n m) with (ble (bS n) m). Setoid-rewriting is not immediate. For instance rewriting (blt n m) $\wedge$ A into (ble (bS n) m) $\wedge$ A for some A requires to have proved that

$$\forall A, B, C, D : \mathsf{Prop}.\ (A \leftrightarrow C) \rightarrow (B \leftrightarrow D) \rightarrow A \wedge B \rightarrow C \wedge D.$$

Such properties can be easily established as far as logical connectives are concerned. We need to be a bit smarter if we consider inductive data type such as sumbool[2]. We state a new theorem

$$\forall A, B, C, D : \mathsf{Prop}.\ (A \leftrightarrow C) \rightarrow (B \leftrightarrow D) \rightarrow \{A\} + \{B\} \rightarrow \{C\} + \{D\}.$$

and use it to prove the conjectures produced by the algorithm presented in section 2.2. It leads to slightly more complicated proofs but enables us to avoid using an axiom in our proof development.

6 Implementation

6.1 Our Tool

We developed a prototype tool. It consists in a ML module which can be plugged into the Coq proof assistant. With this tool, one can build a formal correspondence between two theories. Once data structures and functions in the two settings are defined and once the correspondence between them is formally established, one can automatically translate statements and proof terms to the new representation. The tool, as well as the whole development we carried out, are available online: http://www-sop.inria.fr/lemme/Nicolas.Magaud/Changes/.

6.2 Case Studies

We present the main results of two case studies we carried out. In addition to changing representation for natural numbers, we also experiment with changing representation for polymorphic lists.

Peano's Numbers into Binary Numbers. Natural numbers are our basic example for changes of data structures. The tool we develop allows us to translate the whole library *Arith* of the standard Coq distribution. We first define counterparts of all basic operations (plus, minus, mult...) and relations (le, lt...) on the binary representation of natural numbers by hand. In particular, we prove the dependent recursion operators for nat and their reduction rules as equations.

[2] `Inductive sumbool [A : Prop; B : Prop] : Set :=`
`left : A->A+B | right : B->A+B.`

In addition, we transfer the induction principle for le into the binary setting. Once this part is achieved by the user (i.e reduction rules (ι or δ-reduction) for the newly defined operations are proven as equations), it is straightforward to translate automatically all statements and their proof terms from the unary representation to the binary representation of integers.

It makes it possible to prove all properties required to show the data type bin has the properties of a semi-ring. Therefore we can extend the *Ring* tactic to prove equations in the binary setting directly.

Lists into Reversed Lists. We also study the library *PolyList* of Coq. We transform polymorphic lists into polymorphic reverse lists, defined as an inductive data type rlist. We define counterparts of the functions append, head, tail, and length on reverse lists and proved their properties. In addition, we translated the inductive principle list_ind into an equivalent principle for the reverse lists; we also proved the translated counterparts of the minimal recursion operators for lists as well as their associated reduction rules as equations.

7 Discussion

In this paper, we presented some new techniques to enhance proof reuse when changing data representation in the Coq proof assistant. We improve the approach presented in [12]. It leads to a new tool that appears to be efficient and generic. Indeed, it was usable to translate the whole *Arith* library of Coq. In addition, we manage to reuse it successfully for translating polymorphic lists into reverse polymorphic lists.

Among future research directions, we can quote changing the representation of the index type of an inductive family [8]. For instance, one can imagine to translate dependent lists indexed by nat into dependent lists indexed by bin.

```
Inductive vect [A : Set]  : nat->Set :=
      vnil : (vect A (0))
    | vcons : (n:nat)A->(vect A n)->(vect A (S n))
```

As (pred (S n)) and n are convertible in nat, this statement is well-formed:

$$\forall v : (\mathsf{vect}\ A\ n).\ \forall v' : (\mathsf{vect}\ A\ (\mathsf{pred}\ (\mathsf{S}\ n))).\ v =_{(\mathsf{vect}\ A\ n)} v'$$

However in the binary setting, (pred (S n)) and n will not necessarily be convertible. Consequently, two terms that were propositionally equal in the initial setting do not even live in the same type after the transformation.
A solution (out of scope within the Coq system) would be to identify judgemental (conversion) and propositional equalities, as described in [10]. Another approach may consist in connecting together the two instances of the dependent family via some coercions [16]. Z. Luo and S. Soloviev proposed in [11] some mechanisms to add coercions between a type (e.g. lists) and a family of types (e.g. vectors of length n).

Another direction is to study changes of data type where input and output types are not isomorphic. For instance, one may want to translate lists into trees. Trees would be a more efficient data structure for computing, but different trees can have the same representation as lists. Therefore connecting these two representations may not be so obvious.

References

1. A. Balaa and Y. Bertot. Fix-point Equations for Well-Founded Recursion in Type Theory. In *Theorem Proving in Higher Order Logics: TPHOLs 2000*, volume 1869 of *LNCS*, pages 1–16. Springer-Verlag, 2000.
2. G. Barthe and O. Pons. Type Isomorphisms and Proof Reuse in Dependent Type Theory. In F. Honsell and M. Miculan, editors, *FOSSACS'01*, volume 2030, pages 57–71. LNCS, Springer-Verlag, 2001.
3. Y. Bertot and P. Casteran. *Le Coq'Art*. To appear, 2003.
4. J. Chrzaszcz. Implementation of Modules in the Coq System. Draft, February 2003.
5. Coq development team, INRIA and LRI. *The Coq Proof Assistant Reference Manual*, May 2002. Version 7.3.
6. C. Cornes and D. Terrasse. Automatizing Inversion of Inductive Predicates in Coq. In *TYPES'95*, volume 1158. LNCS, Springer-Verlag, 1995.
7. D. Delahaye and M. Mayero. `Field`: une procédure de décision pour les nombres réels en `Coq`. In P. Castéran, editor, *Journées Francophones des Langages Applicatifs, Pontarlier*. INRIA, Janvier 2001.
8. P. Dybjer. Inductive Families. *Formal Aspects of Computing*, 6(4):440–465, 1994.
9. P. Dybjer. A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory. *Journal of Symbolic Logic*, 65(2), 2000.
10. M. Hofmann. Conservativity of Equality Reflection over Intensional Type Theory. In *BRA Workshop on Types for Proofs and Programs (TYPES'95)*, volume 1158, pages 153–165. Springer-Verlag LNCS, 1996.
11. Z. Luo and S. Soloviev. Dependent Coercions. In *8th Inter. Conf. on Category Theory in Computer Science (CTCS'99)*, volume 29 of *ENTCS*, pages 23–34, Edinburgh, Scotland, 1999. Elsevier.
12. N. Magaud and Y. Bertot. Changing Data Structures in Type Theory:A Study of Natural Numbers. In *International Workshop on Types for Proofs and Programs*, volume 2277 of *LNCS*, pages 181–196. Springer-Verlag, 2000.
13. C. McBride and J. McKinna. The View from the Left. *Journal of Functional Programming*, Special Issue: Dependent Type Theory meets Programming Practice, 2002. submitted.
14. A. Miquel. Axiom $\forall P, Q : Prop.$ (P $\leftrightarrow$ Q)$\rightarrow$(P==Q) is safe. Communication in the coq-club list, November 2002.
15. C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, 1993. LIP research report 92–49.
16. A. Saïbi. Typing Algorithm in Type Theory with Inheritance. In *POPL'97*. ACM, 1997.
17. P. Wadler. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *POPL'87*. ACM, 1987.