Type Isomorphisms and Proof Reuse in Dependent Type Theory

Gilles Barthe[1] and Olivier Pons[2]

[1] INRIA Sophia-Antipolis, France Gilles.Barthe@inria.fr
[2] Departamento de Informática, Universidade do Minho, Portugal
olivier@lmf.di.uminho.pt

Abstract. We propose a theoretical foundation for proof reuse, based on the novel idea of a computational interpretation of type isomorphisms.

1 Introduction

Background Proof development systems based on dependent type theory, such as Coq [4], Lego [29] and PVS [38], are increasingly used for large-scale formalisations of mathematics and programming languages semantics. These formalisations are usually available on-line and, in some instances, form an integral part of the system's distribution. In principle, these formalisations should allow users to benefit from a large library of basic results upon which to build up their own developments. In practice, users often face difficulties in adapting existing formalisations to their problem, leading to duplicate work and un-integrated contributions. Thus it is important to provide techniques and tools that facilitate proof reuse. However, existing techniques, which we detail below, fall short of providing a satisfactory answer to the problem.

Type isomorphisms provide a type-theoretical notion of equivalence between (generally simple) types: in a nutshell, a type isomorphism between A and B is a pair of expressions $f : A \to B$ and $g : B \to A$ that are mutually inverse to each other (usually w.r.t. $\beta\eta$-convertibility). Type isomorphisms have been used for a number of purposes, including program reuse [21, 22, 34, 35], but the potential of type isomorphisms in proof development systems remains largely unexplored.

This article The purpose of this paper is two-fold:

1. to highlight a number of situations where (a computational interpretation of) type isomorphisms can be used to good effect for proof reuse. More precisely, we suggest how, in the context of dependent type theory, type isomorphisms (1) enhance the usability of generic frameworks (2) allow to switch between equivalent representations of mathematical theories or data structures. En passant we also show how type isomorphisms are useful in the context of programming with dependent types;

2. to propose an extension of dependent type theory where type isomorphisms are given a prominent rôle through computational rules. More precisely, we define an extension of dependent type theory in which type isomorphisms are captured by rewriting rules at the levels of types and elements. We show that the extension is well-behaved, i.e. enjoys all standard properties required for proof-checking, including subject reduction, consistency and decidable type-checking (modulo strong normalisation, which is not proved here).

Our broad conclusion is that type isomorphisms are useful in a number of situations (mostly connected to proof reuse) where traditional dependent type theory and its extensions are too weak.

Before proceeding any further, let us dispose of a red herring: in general, type isomorphisms are studied in extensional type theories and their existence is undecidable. Here we do not need extensionality because we enforce isomorphisms through new rewrite rules. As to undecidability, it simply reflects itself in the fact that our rewrite rules are not "complete ", and that it is possible to introduce new rewrite rules that preserve the good properties of the calculus.

Related work Our work is original in that it suggests, for the first time to our best knowledge, to give a computational interpretation to (a class of) type isomorphisms. However, our work fits in a series of efforts to lay theoretical foundations for proof reuse in dependent type theory.

Subtyping Subtyping [16, 17] is a basic mechanism to enhance proof reuse in type systems. Specific forms of subtyping connected to reusability include record subtyping [11] and constructor subtyping [8] but these forms of subtyping are too specialised to provide a general solution to the problem of reusability.

Implicit coercions Implicit coercions [3, 5, 28, 36] provide a very powerful mechanism that subsumes a variety of previous and seemingly unrelated proposals in the area of subtyping. In a nutshell, a coercion from A to B is a function $f : A \to B$, declared in a *coercion context*, and which allows to view a as an element of B and a shorthand for $f\ a$ whenever $a : A$. While the initial idea is relatively simple, its realization involves a number of difficult concepts:

- coherence of coercions: a set of coercions is coherent if the graph of coercions it generates, notably by composition and instantiation, is such that, for every types A and B, any two edges (coercions) from A to B are convertible. Coherence is undecidable in general, and we currently lack of criteria to decide whether a set of coercions is indeed coherent. Thus current implementations of coercive subtyping, and in particular [4, 13], do not check coherence to the detriment of conceptual clarity;
- computational interpretation of coercions: in most existing works, coercions are not given a computational meaning through rewrite rules. Instead, the meaning of coercive subtyping is given in a logical framework with equality judgements, as in [28], or by a translation of the extended language with

coercions into the original language without coercions, as in [3, 36]. We prefer to follow the usual approach where expressions are given a computational interpretation independently of typing or of another system;
- back-and-forth coercions: the examples of this paper involve back-and-forth coercions, i.e. pairs of coercions $f : A \to B$ and $g : B \to A$ that coexist in a single coercion context. However, most works on implicit coercions do not consider such back-and-forth coercions. Moreover, in case back-and-forth coercions are considered the coercions f and g are required to be mutually inverse, which is not the case for our examples, as we work in a type theory without extensionality (η-conversion).

Thus our work may be viewed as contributing to the understanding of implicit coercions by suggesting (1) the usefulness of back-and-forth coercions (2) a novel computational interpretation of coercions.

Proof transformation While the approaches presented above involve modifying the type system, several authors have developed methodologies to modify and adapt proofs to another context. Existing techniques include proof by analogy, see e.g. [20], proof by generalisation, see e.g. [23, 25, 27, 33], or proof by transformation, see e.g. [30]. These solutions are appealing in that they do not involve modifying the type system and can address a wealth of problems that range beyond the issues considered in this paper. However, these approaches are not always implemented, can be heavy to put in practice and/or can yield large proof terms.

Type-theoretical preliminaries The type theory considered in this paper is essentially the Calculus of Inductive Constructions, see e.g. [39]. We use **Prop**, **Set** and **Class** to denote the universes of sets, propositions and classes respectively—usually these are called **Prop**, $\mathbf{Type}_0$ and $\mathbf{Type}_1$ respectively. We let $\triangle$ range over universes.

In addition, we use $\mathbb{N}$ to denote the inductive type of natural numbers (with constructors 0 and S). Besides, we use $\{\mathsf{l} : A,\ \mathsf{l'} : A'\}$ for denoting record types, $\{\mathsf{l} = a,\ \mathsf{l'} = a'\}$ for denoting records, the standard dot notation to denote field selection and $.\langle . := . \rangle$ to denote substitution. Finally, we let $\doteq$ denote Leibniz equality.

We conclude with a warning on naming: most of the type isomorphisms introduced in this paper involve the generation of new constructor names for inductive types or label names for records. In this paper, we gloss over the way such labels can be generated and fixed uniquely.

Contents The remaining of the paper is organised as follows: in Section 2, we illustrate a number of situations where type isomorphisms can be used for proof reuse. For each example, we suggest an extension of the type theory. Properties of the extensions are discussed in Section 3. Finally, we conclude in Section 4.

2 Motivating examples

In this section, we introduce a number of examples where a computational interpretation of type isomorphisms allows users to switch between different representations of the same object/structure. Before delving into the examples, let us emphasise that the first three examples arise from our previous and ongoing work on the formalisation of mathematics and programming language semantics, and that in all cases, we were overwhelmed by the burden of switching between existing representations manually.

2.1 Universal algebra

Universal algebra, see e.g. [18], is a generic framework for the study of algebraic structures. As such, it provides an appealing structuring mechanism for developing a large body of algebra in type theory. Unfortunately, instantiating universal algebra to a concrete algebraic theory is problematic. Definitions are only isomorphic, but not convertible to the ones we would expect. Below we show how type isomorphisms solve the problem.

Setting In this paragraph, we briefly review how to formalise (some representative notions of) universal algebra in type theory. Recall that the framework is parametrised over the notion of signature. Informally, a signature consists of a set F of function symbols, and of a map $\mathsf{ar} : F \to \mathbb{N}$ which assigns its arity to each function symbol.

Definition 1. The type SIGNATURE of *signatures* is defined by

$$\textsc{Signature} : \mathbf{Class} = \{\mathsf{fun} : \mathbf{Set},\ \mathsf{ar} : \mathsf{fun} \to \mathbb{N}\}$$

Our representation is the standard one but there are alternatives formalisations that only focus on finite signatures, see e.g. [14].

Each signature has an associated notion of algebra. Recall that an algebra for a signature $S = (F, \mathsf{ar})$ consists of a set A, called the carrier of the algebra, and of a function $f : A^{\mathsf{ar}(f)} \to A$ for every $f \in F$. To formalise algebras, we therefore need to formalise sets and n-ary functions. For the sake of simplicity, we formalise sets as types and not as setoids [7]. The type of n-ary functions over a type A is denoted by FUN n A and is defined by the recursive equation

$$\begin{aligned}\textsc{Fun}[n : \mathbb{N}][A : \mathbf{Set}]\ &:\ \mathbf{Set}\\ &= \mathsf{case}\ n\ \mathsf{of}\quad 0\ \Rightarrow A\\ &\qquad\qquad \mid\ S\,m \Rightarrow A \to (\textsc{Fun}\ m\ A)\end{aligned}$$

Definition 2. The type ALGEBRA of *algebras over a signature* is defined by

$$\begin{aligned}\textsc{Algebra}[S : \textsc{Signature}]\ &:\ \mathbf{Class}\\ &= \{\mathsf{el} : \mathbf{Set},\ \mathsf{int} : \Pi f : (S \cdot \mathsf{fun}).\ \textsc{Fun}\ (S \cdot \mathsf{ar}\ f)\ \mathsf{el}\}\end{aligned}$$

The formalisation may be developed further, for example by introducing the notion of substructure, quotient... Finally some techniques such as reflection, see e.g. [9], require to define the type of terms of a signature.

Definition 3. The type TERM of *terms* is defined as

$$\begin{array}{l}\textbf{Inductive}\ \textsc{Term}[S : \textsc{Signature}] : \textbf{Set} \\ \quad = \mathit{vterm} : \mathbb{N} \to (\textsc{Term}\ S) \\ \quad \mid \mathit{fterm} : \Pi f : (S \cdot \mathsf{fun}).\ (\textsc{Fun}\ (S \cdot \mathsf{ar}\ f)\ (\textsc{Term}\ S))\end{array}$$

Note that the type of *fterm* does not fit the official definition of constructor type but it is safe to extend the definition of constructor type for the above definition to be considered as legal (see [15] for an account of a similar mechanism). An alternative would be to use the type $(\textsc{Vec}\ (S \cdot \mathsf{ar}\ f)\ (\textsc{Term}\ S)) \to (\textsc{Term}\ S)$ instead of $\textsc{Fun}\ (S \cdot \mathsf{ar}\ f)\ (\textsc{Term}\ S)$ but it would introduce unnecessary technicalities in our presentation.

Problem Now assume that we want to instantiate the framework to a specific signature, say the signature of groups defined below.

Definition 4. The signature GROUPSIG is defined by

$$\textsc{GroupSig} = \{\mathsf{fun} = \textsc{GroupSym},\ \mathsf{ar} = \textsc{GroupAr}\}$$

where

$$\begin{array}{l}\textbf{Inductive}\ \textsc{GroupSym} :\ \textbf{Set} \\ \qquad = \textsc{o} : \textsc{GroupSym} \mid \textsc{e} : \textsc{GroupSym} \mid \textsc{i} : \textsc{GroupSym}\end{array}$$

and

$$\begin{array}{l}\textsc{GroupAr}[f : \textsc{GroupSym}] :\ \mathbb{N} \\ \qquad = \mathsf{case}\ f\ \mathsf{of}\ \textsc{o} \Rightarrow 2 \\ \qquad\quad \mid \textsc{e} \Rightarrow 0 \\ \qquad\quad \mid \textsc{i} \Rightarrow 1\end{array}$$

The type of group algebras, i.e. ALGEBRA GROUPSIG, may be evaluated to

$$\{\mathsf{el} : \textbf{Set},\ \mathsf{int} : \Pi f : \textsc{GroupSig}.\ \textsc{Fun}\ (\textsc{GroupAr}\ f)\ \mathsf{el}\}$$

and there is no further way to proceed. This is bad news, as one would prefer to use the more standard and palatable representation

$$\{\mathsf{el} : \textbf{Set},\ \mathsf{oint} : \mathsf{el} \to \mathsf{el} \to \mathsf{el},\ \mathsf{eint} : \mathsf{el}, \mathsf{iint} : \mathsf{el} \to \mathsf{el}\} \qquad (*)$$

Similarly, the type of group terms, i.e. TERM GROUPSIG, may be evaluated to

$$\begin{array}{l}\textbf{Inductive}\ \textsc{TermGroupSig} : \textbf{Set} \\ \quad = \mathit{vtermGroupSig} : \mathbb{N} \to \textsc{TermGroupSig} \\ \quad \mid \mathit{ftermGroupSig} : \Pi f : \textsc{GroupSym}.\ \textsc{Fun}\ (\textsc{GroupAr}\ f)\ \textsc{TermGroupSig}\end{array}$$

(observe the renaming of constructors, over which we gloss here) whereas one would prefer to use the more standard and palatable representation

$$\begin{array}{l}\textbf{Inductive}\ \textsc{TermGroup} : \textbf{Set} \\ \quad = \mathit{vtermgroup} : \mathbb{N} \to \textsc{TermGroup} \\ \quad \mid \mathit{ftermgroupo} : \textsc{TermGroup} \to \textsc{TermGroup} \to \textsc{TermGroup} \\ \quad \mid \mathit{ftermgroupe} : \textsc{TermGroup} \\ \quad \mid \mathit{ftermgroupi} : \textsc{TermGroup} \to \textsc{TermGroup}\end{array}$$

Solution One may achieve a simple solution to the problem by forcing suitable type isomorphisms. Indeed, consider the type of group algebras. The type

$$\Pi f : \textsc{GroupSig}.\ \textsc{Fun}\ (\textsc{GroupAr}\ f)\ A$$

is inhabited by functions over the 3-elements type GroupSig. Now, such functions can alternatively be described by triples assigning a value to each element of GroupSig, i.e. by inhabitants of the type

$$\{\mathsf{oint} : \textsc{Fun}\ (\textsc{GroupAr}\ \textsc{o})\ A,\ \mathsf{eint} : \textsc{Fun}\ (\textsc{GroupAr}\ \textsc{e})\ A, \mathsf{iint} : \textsc{Fun}\ (\textsc{GroupAr}\ \textsc{i})\ A\}$$

which reduces to

$$\{\mathsf{oint} : A \to A \to A,\ \mathsf{eint} : A, \mathsf{iint} : A \to A\}$$

The approach advocated in this paper is to enforce an equational rule that relates dependent function spaces over an enumeration type to record types that collect the value of the function for each element of the enumeration type. More precisely, if X_n is an enumeration type with inhabitants $\bullet_1, \ldots, \bullet_n$, then we introduce the equational rule

$$\Pi f : \mathrm{X}_n.\ T \quad =_\Sigma \quad \{\mathsf{lab}_1 : T\langle f := \bullet_1\rangle,\ \ldots\ ,\ \mathsf{lab}_n : T\langle f := \bullet_n\rangle\}$$

for some previously agreed upon set of labels $\mathsf{lab}_1,\ \ldots\ ,\ \mathsf{lab}_n$. This equational rule is integrated into the conversion rule so as to allow to switch between the two representations. As a result, we introduce non-canonical inhabitants at function and record types. For example, Σ-conversion makes it possible to apply a record to an argument, or to select a field of a function! For example, taking G : Algebra GroupSig one can type $(G \cdot \mathsf{int})\ \textsc{o}$, whereas $(G \cdot \mathsf{int})$ is of a record type. We therefore need to introduce new computational rules for such cases

$$\begin{array}{r}\{\mathsf{lab}_1 = f_1,\ \ldots\ , \mathsf{lab}_n = f_n\}\ \bullet_i \to_\sigma f_i \\ (\lambda f : \mathrm{X}_n.e) \cdot \mathsf{lab}_i \to_\sigma e\langle f := \bullet_i\rangle\end{array}$$

Coming back to the definition of group algebras, the new reduction relation enforces the type Algebra GroupSig to reduce to

$$\{\mathsf{el} : \textbf{Set},\ \mathsf{int} : \{\mathsf{oint} : \mathsf{el} \to \mathsf{el} \to \mathsf{el},\ \mathsf{eint} : \mathsf{el}, \mathsf{iint} : \mathsf{el} \to \mathsf{el}\}\}$$

which closely resembles $(*)$: in fact, one obtains $(*)$ simply by removing the nesting of records. In the next example, we show how new computational rules

allow to flatten records, thus forcing the two above types to be convertible with the usual representation of group algebras.

Let us now go back to the type of group terms. By applying the Σ-rule suggested above, one can rewrite TermGroupSig to

$$\begin{array}{l}\textbf{Inductive}\ \textsc{TermGroupSig} : \textbf{Set}\\ \quad = \mathit{vtermGroupSig} : \mathbb{N} \to \textsc{TermGroupSig}\\ \quad \mid \mathit{ftermGroupSig} :\\ \quad\quad \{\ \mathsf{oint} : \textsc{TermGroupSig} \to \textsc{Term GroupSig} \to \textsc{TermGroupSig},\\ \quad\quad\ \mathsf{eint} : \textsc{TermGroupSig},\\ \quad\quad\ \mathsf{iint} : \textsc{TermGroupSig} \to \textsc{TermGroupSig}\}\end{array}$$

Clearly, the above expression is ill-formed as the type of *ftermGroupSig* does not fit the definition of constructor type. However, it is easy to recover a well-formed definition, and in fact the expected one, by splitting the second constructor into three. Formally, we adopt the compatibility rule:

$$\frac{T =_\Sigma \{\mathsf{lab}_1 : T_1,\ \ldots\ , \mathsf{lab}_n : T_n\}}{\begin{array}{c}\textbf{Inductive}\ I : A = \ \ldots \mid c : T \mid \ \ldots\\ =_\Sigma\\ \textbf{Inductive}\ I : A = \ \ldots \mid c_1 : T_1 \mid \ \ldots \mid c_n : T_n \mid \ \ldots\end{array}}$$

instead of the rule of compatible closure of Σ-equality for inductive types.

2.2 Switching between mathematical theories

In a recent article [32], R. Pollack surveys the different existing mechanisms to represent mathematical structures in type theory. Record types are one well-known such mechanism. However, as pointed out in *loc. cit.*, record types come in several flavours, including left-associating and right-associating records. All of these approaches are extensively used in the formalisation of mathematics.

Setting We briefly compare left-associating and right-associating records in the context of Partial Equivalence Relations (PERs). PERs may be defined as flat, right-associative or left-associative records.

Definition 5.

1. The *flat construction of* Per can be defined by

$$\begin{array}{l}\textsc{Per} : \textbf{Class}\\ \quad = \{\mathsf{S} : \textbf{Set},\ \mathsf{R} : \mathsf{S} \to \mathsf{S} \to \textbf{Prop},\ \mathsf{Sym} : (\textsc{sym}\ \mathsf{S}\ \mathsf{R}),\ \mathsf{Trans} : (\textsc{trans}\ \mathsf{S}\ \mathsf{R})\}\end{array}$$

2. The *right-associative construction of* Per is defined by

$$\begin{array}{rl}\textsc{Inner}[S : \textbf{Set}][R : S \to S \to \textbf{Prop}] : \textbf{Prop} & \\ \quad = \{\mathsf{Sym} : (\textsc{sym}\ S\ R),\ \mathsf{Trans} : (\textsc{trans}\ S\ R)\} & \\ \textsc{Middle}[S : \textbf{Set}] : \textbf{Set} = \{\mathsf{R} : S \to S \to \textbf{Prop},\ \mathsf{i} : (\textsc{Inner}\ S\ \mathsf{R})\} & \\ \textsc{Per}_r : \textbf{Class} = \{\mathsf{S} : \textbf{Set},\ \mathsf{r} : \textsc{Middle}\ \mathsf{S}\} & \end{array}$$

3. The *left-associative construction of* PER is defined by

$$\begin{aligned}\textsc{Relation} &: \mathbf{Class} = \{\mathsf{S} : \mathbf{Set},\ \mathsf{R} : \mathsf{S} \to \mathsf{S} \to \mathbf{Prop}\}\\ \textsc{SymRel} &: \mathbf{Class} = \{\mathsf{P} : \textsc{Relation},\ \mathsf{Sym} : \textsc{sym}\ (\mathsf{P} \cdot \mathsf{S})\ (\mathsf{P} \cdot \mathsf{R})\}\\ \textsc{Per}_l &: \mathbf{Class} = \{\mathsf{SR} : \textsc{SymRel},\ \mathsf{Trans} : \textsc{trans}\ (\mathsf{SR} \cdot \mathsf{P} \cdot \mathsf{S})\ (\mathsf{SR} \cdot \mathsf{P} \cdot \mathsf{R})\}\end{aligned}$$

Problem As noted by R. Pollack, all approaches have their own advantages and drawbacks. Right-associative constructions are easy to specialise but hard to extend. Conversely, left-associative constructions are easy to extend but hard to specialise. Both specialisation and extensibility are important and thus all approaches are extensively used in practice. It is therefore important to be able to switch between the three representations.

Solution One may solve the problem by enforcing suitable equational rules that flatten record types. In order to avoid mind-boggling renaming problems with labels, we formalise these rules using Coq's view of record types as inductive types with a single constructor. The rule then becomes a

$$\begin{aligned}&\mathbf{Inductive}\ I : \triangle = c : \Pi \boldsymbol{x}{:}\boldsymbol{A}.\ \Pi y{:}(\mathbf{Inductive}\ J : \triangle' = d : \Pi \boldsymbol{z}{:}\boldsymbol{B}.\ J).\ \Pi \boldsymbol{x'}{:}\boldsymbol{A'}.\ I\\ =_\Sigma\ &\mathbf{Inductive}\ I' : \triangle = c' : \Pi \boldsymbol{x}{:}\boldsymbol{A}.\ \Pi \boldsymbol{z}{:}\boldsymbol{B}.\ \Pi \boldsymbol{x'}{:}\boldsymbol{A'}\langle \mathrm{y} := d\ \boldsymbol{z}\rangle.\ I'\end{aligned}$$

As before the equational rule at the type level needs to be accompanied by rewrite rules at the object level. The rules allow to reduce case-expressions whose argument is not in the right shape.

$$\begin{aligned}&\mathsf{case}\ (c\ \boldsymbol{M}\ (d\ \boldsymbol{N})\ \boldsymbol{M'})\ \mathsf{of}\ \{c' \Rightarrow e\} \to_\sigma \mathsf{case}\ (c'\ \boldsymbol{M}\ \boldsymbol{N}\ \boldsymbol{M'})\ \mathsf{of}\ \{c' \Rightarrow e\}\\ &\mathsf{case}\ (c'\ \boldsymbol{M}\ \boldsymbol{N}\ \boldsymbol{M'})\ \mathsf{of}\ \{c \Rightarrow e\} \to_\sigma \mathsf{case}\ (c\ \boldsymbol{M}\ (d\ \boldsymbol{N})\ \boldsymbol{M'})\ \mathsf{of}\ \{c \Rightarrow e\}\end{aligned}$$

Let us return to the example of PERs and observe the behaviour of the rewrite rules for this example. We focus on left-associating representations, since they are more intricate to handle. Formalising records as inductive types, the left-associating definition of PERs becomes

$$\begin{aligned}&\mathbf{Inductive}\ \textsc{Relation} : \mathbf{Class}\\ &= c_{Rel} : \Pi S : \mathbf{Set}.\ (S \to S \to \mathbf{Prop}) \to \textsc{Relation}\\ &\mathbf{Inductive}\ \textsc{SymRel}\ \ : \mathbf{Class}\\ &= c_{SymRel} : \Pi P : \textsc{Relation}.\ (\textsc{sym}\ (\textsc{Car}\ P)\ (\textsc{Rel}\ P)) \to \textsc{SymRel}\\ &\mathbf{Inductive}\ \textsc{Per}_l\ \ \ \ : \mathbf{Class}\\ &= c_{Perl} : \Pi SR : \textsc{SymRel}.\ (\textsc{trans}\ (\textsc{Car}\ (\mathrm{RSR}\ SR))\ (\textsc{Rel}\ (\mathrm{RSR}\ SR))) \to \textsc{Per}_l\end{aligned}$$

where

$$\begin{aligned}\textsc{Car} &: \textsc{Relation} \to \mathbf{Set}\\ \textsc{Rel} &: \Pi P : \textsc{Relation}.\ (\textsc{Car}\ P) \to (\textsc{Car}\ P) \to \mathbf{Prop}\\ \mathrm{RSR} &: \textsc{SymRel} \to \textsc{Relation}\end{aligned}$$

are defined in the obvious way. By using Σ-conversion as well as $\beta\iota\beta^+$-reduction, we can derive the flat representation of PERs

$$\begin{aligned}&\mathbf{Inductive}\ \textsc{Per'}_l : \mathbf{Class}\\ &\quad = c_{Perl'} : \Pi S : \mathbf{Set}.\ \Pi R : S \to S \to \mathbf{Prop}.\\ &\qquad\qquad (\textsc{Sym}\ S\ R) \to (\textsc{Trans}\ S\ R) \to \textsc{Per}'_l\end{aligned}$$

2.3 Switching between equivalent data structures

Standard data structures, such as natural numbers or lists, often have several representations. In many cases, these representations serve different purposes: typically, some of them will be well-suited for logical reasoning whereas some others will be well-suited to serve as a basis for efficient algorithms. Below we illustrate some of the issues involved by considering natural numbers, and their representation in unary and binary schemes. Both representations are used for example in J. C. B. Almeida's formalisation of the RSA algorithm [1].

Setting Natural numbers are traditionally formalised in type systems with the Peano unary representation scheme.

Definition 6. The type $\mathbb{N}$ of unary natural numbers is defined by

$$\begin{aligned}\textbf{Inductive}\ \mathbb{N}\ : \textbf{Set} = {} & 0 : \mathbb{N} \\ \mid {} & S : \mathbb{N} \to \mathbb{N}\end{aligned}$$

One can also use an alternative binary representation scheme in which a natural number is encoded as 0, $2n+1$ or $2n+2$ (this scheme slightly differs from the Coq encoding, which distinguishes between 0 and positive numbers).

Definition 7. The type $\mathbb{B}$ of binary natural numbers is defined by

$$\begin{aligned}\textbf{Inductive}\ \mathbb{B} : \textbf{Set} = {} & B_H : \mathbb{B} \\ \mid {} & B_O : \mathbb{B} \to \mathbb{B} \\ \mid {} & B_I : \mathbb{B} \to \mathbb{B}\end{aligned}$$

The two representations may be related by translations in both directions. These translations are based on encodings of a datatype's constructors as functions on the other datatype.

Definition 8.

1. The successor function $\mathrm{S}^{\mathbb{B}}$ is defined by the recursive equation

$$\begin{aligned}\mathrm{S}^{\mathbb{B}}[b : \mathbb{B}] : \mathbb{B} = {} & \mathsf{case}\ b\ \mathsf{of} \quad B_H \Rightarrow B_O\ B_H \\ \mid {} & B_O\ c \Rightarrow B_I\ c \\ \mid {} & B_I\ c \Rightarrow B_O\ (\mathrm{S}^{\mathbb{B}}\ c)\end{aligned}$$

2. The conversion function $\textsc{n2b}$ is defined by the recursive equation

$$\begin{aligned}\textsc{n2b}[n : \mathbb{N}] : \mathbb{B} = {} & \mathsf{case}\ n\ \mathsf{of} \quad 0 \Rightarrow B_H \\ \mid {} & S\ m \Rightarrow \mathrm{S}^{\mathbb{B}}\ (\textsc{n2b}\ m)\end{aligned}$$

3. The function $\mathrm{B}_O^{\mathbb{N}}$ is defined as

$$\mathrm{B}_O^{\mathbb{N}}[n : \mathbb{N}] : \mathbb{N} = S\ (\textsc{Twice}\ n)$$

where $\textsc{Twice}$ is the multiplication by 2 over $\mathbb{N}$.

4. The function $\mathrm{B}_I^{\mathbb{N}}$ is defined as

$$\mathrm{B}_I^{\mathbb{N}}[n : \mathbb{N}] : \mathbb{N} = \textsc{Twice}\ (S\ n)$$

5. The conversion function B2N is defined by the recursive equation

$$\begin{array}{l}\textsc{b2n}[b : \mathbb{B}] : \mathbb{N} := \ \mathsf{case}\ b\ \mathsf{of}\quad B_H \Rightarrow 0 \\ \qquad\qquad\qquad\qquad\qquad\quad \mid\ B_O\ c \Rightarrow B_O^{\mathbb{N}}\ (\textsc{b2n}\ c) \\ \qquad\qquad\qquad\qquad\qquad\quad \mid\ B_I\ c \Rightarrow B_I^{\mathbb{N}}\ (\textsc{b2n}\ c)\end{array}$$

Problem We would like to use interchangeably both representations of natural numbers. In particular, we would like to derive the correctness of RSA on binary numbers from the correctness of RSA on unary numbers.

Solution One may solve the problem by forcing the types $\mathbb{N}$ and $\mathbb{B}$ to be convertible by adding a new Σ-conversion rule

$$\mathbb{N} \quad =_\Sigma \quad \mathbb{B}$$

The new conversion rule introduces non-canonical inhabitants at each type. For example, it makes it possible to do a $\mathbb{N}$-case analysis on an inhabitant of $\mathbb{B}$ and vice-versa! We therefore need to cater for such cases, by introducing new computational rules which avoid "stuck redexes", in this case a case-expression applied to a term whose head symbol is a constructor of the "wrong" datatype

$$\begin{array}{l}\mathsf{case}\ F\ \mathsf{of}\ \{E_N\} \rightarrow_\sigma \mathsf{case}\ (\textsc{b2n}\ F)\ \mathsf{of}\ \{E_N\} \\ \mathsf{case}\ G\ \mathsf{of}\ \{E_B\} \rightarrow_\sigma \mathsf{case}\ (\textsc{n2b}\ G)\ \mathsf{of}\ \{E_B\}\end{array}$$

where F is either B_H, $B_O\ M$ or $B_I\ M$ and G is either 0 or $S\ M$ and

$$\begin{array}{l}E_N = 0 \Rightarrow f_0 \mid S\ x \Rightarrow f_s \\ E_B = B_H \Rightarrow f_H \mid B_O\ x \Rightarrow f_O \mid B_I\ x \Rightarrow f_I\end{array}$$

Let us now turn to closed terms in normal form. For every natural number n, its unary and binary encodings $\lceil n \rceil^{\mathbb{N}}$ and $\lceil n \rceil^{\mathbb{B}}$ are normal closed inhabitants of $\mathbb{N}$ and $\mathbb{B}$. Using the equality $(\mathsf{case}\ x\ \mathsf{of}\ 0 \Rightarrow 0 \mid S\ x \Rightarrow S\ x) \doteq x$, one can prove that $\lceil n \rceil^{\mathbb{N}} \doteq \lceil n \rceil^{\mathbb{B}}$ for every natural number n. Thus, our calculus does not have the so-called equality reflection property. This property may be recovered by introducing further equational rules $\lceil n \rceil^{\mathbb{N}} \leftrightarrow_\varsigma \lceil n \rceil^{\mathbb{B}}$. Embedding these equational rules in the conversion rule of the type system does not affect the decidability of type-checking.

2.4 Expressiveness of programming languages with dependent types

Our last example, which is not connected to proof reuse, shows how type isomorphisms can also be used in situations where the standard convertibility relation is too weak. Incidentally the following example, which deals with dependent lists (a.k.a. vectors), shows how a dependently typed programming language such as Cayenne [2], a dialect of Haskell that extends the Calculus of Constructions with full recursion, could be strengthened by using features from DML [40], a dependently typed extension of ML with a restricted form of dependent types.

Setting Recall that vectors are formalised via the family $\textsc{vec} : \mathbb{N} \to \mathbf{Set} \to \mathbf{Set}$ which associates to every natural number n and type A the type of vectors of elements of A with length n. One can define vector concatenation and vector inversion with the types

$$\textsc{app} : \Pi m, n : \mathbb{N}.\ \Pi A : \mathbf{Set}.\ (\textsc{vec}\ m\ A) \to (\textsc{vec}\ n\ A) \to (\textsc{vec}\ (\textsc{plus}\ m\ n)\ A)$$
$$\textsc{rev} : \Pi m : \mathbb{N}.\ \Pi A : \mathbf{Set}.\ (\textsc{vec}\ m\ A) \to (\textsc{vec}\ m\ A)$$

The problem In the context

$$m : \mathbb{N},\ l : \textsc{vec}\ m\ A,\ n : \mathbb{N},\ l' : \textsc{vec}\ n\ A$$

one would expect to be able to prove

$$\textsc{rev}\ (\textsc{plus}\ m\ n)\ A\ (\textsc{app}\ m\ n\ A\ l\ l') \doteq \textsc{app}\ n\ m\ A\ (\textsc{rev}\ n\ A\ l')\ (\textsc{rev}\ m\ A\ l)$$

Unfortunately, the above is not well-typed since:

- the left-hand side of the equality has type $\textsc{vec}\ (\textsc{plus}\ m\ n)\ A$;
- the right-hand side of the equality has type $\textsc{vec}\ (\textsc{plus}\ n\ m)\ A$;
- the two types are not convertible in general (since $\textsc{plus}\ m\ n$ and $\textsc{plus}\ n\ m$ are Leibniz equal but not convertible in general).

Solution We impose a type isomorphism through the new equational rule

$$\textsc{vec}\ (\textsc{plus}\ m\ n)\ A \quad =_\Sigma \quad \textsc{vec}\ (\textsc{plus}\ n\ m)\ A$$

Such a use of type isomorphisms does not require the introduction of new computational rules at the term level, intuitively because $\textsc{plus}\ m\ n \doteq \textsc{plus}\ n\ m$ for every $m, n : \mathbb{N}$. We do not develop this example further, since a neat treatment of convertibility-checking and type-checking algorithms for this example requires to treat PLUS as a constant function symbol specified by pattern-matching, see e.g. [12], and thus to extend the type theory even further.

3 The type theory and its properties

3.1 The extended type system

The extended type system is very closely related to the Calculus of Inductive Constructions: it shares the same set of terms, the same notion of substitution, the same notion of constructor type, inductive definition, guarded recursive definitions... The sole difference with Calculus of Inductive Constructions is the computational behaviour of expressions. In addition to β-reduction, β^+-reduction and ι-reduction, which respectively account for the computational interpretation of functions, guarded recursive definitions and case-expressions, we consider the rules of σ-reduction displayed in Figure 1 and the equational rules of Σ-equality displayed in Figure 2; for the latter, special provisions in the compatibility rules are enforced so as to avoid the problems suggested in Subsection 2.1. These are embedded in the conversion rule, which becomes

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B} \quad A =_{\beta\beta^+\iota\sigma\Sigma} B$$

$$
\begin{array}{|r|}
\hline
\{\mathsf{lab}_1 = f_1,\ \ldots\ , \mathsf{lab}_n = f_n\}\ \bullet_i \to_\sigma f_i \\
(\lambda f : \mathrm{X}_n.e) \cdot \mathsf{lab}_i \to_\sigma e\langle f := \bullet_i\rangle \\
\hline
\mathsf{case}\ (c\ \boldsymbol{M}\ (d\ \boldsymbol{N})\ \boldsymbol{M'})\ \mathsf{of}\ \{c' \Rightarrow e\} \to_\sigma \mathsf{case}\ (c'\ \boldsymbol{M}\ \boldsymbol{N}\ \boldsymbol{M'})\ \mathsf{of}\ \{c' \Rightarrow e\} \\
\mathsf{case}\ (c'\ \boldsymbol{M}\ \boldsymbol{N}\ \boldsymbol{M'})\ \mathsf{of}\ \{c \Rightarrow e\} \to_\sigma \mathsf{case}\ (c\ \boldsymbol{M}\ (d\ \boldsymbol{N})\ \boldsymbol{M'})\ \mathsf{of}\ \{c \Rightarrow e\} \\
\hline
\mathsf{case}\ F\ \mathsf{of}\ \{E_N\} \to_\sigma \mathsf{case}\ (\textsc{b2n}\ F)\ \mathsf{of}\ \{E_N\} \\
\mathsf{case}\ G\ \mathsf{of}\ \{E_B\} \to_\sigma \mathsf{case}\ (\textsc{n2b}\ G)\ \mathsf{of}\ \{E_B\} \\
\hline
\end{array}
$$

where $F = B_H \mid B_O\ M \mid B_I\ M$ and $G = 0 \mid S\ M$ and

$$
\begin{array}{l}
E_N = 0 \Rightarrow f_0 \mid S\ x \Rightarrow f_s \\
E_B = B_H \Rightarrow f_H \mid B_O\ x \Rightarrow f_O \mid B_I\ x \Rightarrow f_I
\end{array}
$$

Fig. 1. Rules for σ-reduction

$$
\begin{array}{|r|}
\hline
\Pi f : \mathrm{X}_n.\ T =_\Sigma \{\mathsf{lab}_1 : T\langle f := \bullet_1\rangle,\ \ldots\ ,\ \mathsf{lab}_n : T\langle f := \bullet_n\rangle\} \\
\hline
\mathbb{N} =_\Sigma \mathbb{B} \\
\hline
\mathbf{Inductive}\ I : \triangle = c : \Pi \boldsymbol{x}\!:\!\boldsymbol{A}.\ \Pi y\!:\!(\mathbf{Inductive}\ J : \triangle' = d : \Pi \boldsymbol{z}\!:\!\boldsymbol{B}.\ J).\ \Pi \boldsymbol{x'}\!:\!\boldsymbol{A'}.\ I \\
=_\Sigma\ \ \mathbf{Inductive}\ I' : \triangle = c' : \Pi \boldsymbol{x}\!:\!\boldsymbol{A}.\ \Pi \boldsymbol{z}\!:\!\boldsymbol{B}.\ \Pi \boldsymbol{x'}\!:\!\boldsymbol{A'}\langle \mathrm{y} := d\ \boldsymbol{z}\rangle.\ I' \\
\hline
\end{array}
$$

Fig. 2. Rules for Σ-equalities

3.2 Properties of the extended type system

First, the extended reduction relation is confluent. Indeed, the new σ-reduction rules are formulated in such a way that there are no critical pairs so the reduction relation is orthogonal and hence confluent [26].

Proposition 9. *$\beta\beta^+\iota\sigma$-reduction is confluent.*

Second, the calculus enjoys subject reduction.

Proposition 10. *If $\Gamma \vdash e : A$ and $e \to_{\beta\beta^+\iota\sigma} e'$ then $\Gamma \vdash e' : A$.*

Third, the calculus enjoys decidable type-checking, provided every legal term is strongly normalising. A key argument in the proof is that convertibility between legal types is decidable, see e.g. [6, 31, 37] for a recent survey of type-checking algorithms for dependent type theory.

Proposition 11. *If every legal term is normalising, then it is decidable whether $\Gamma \vdash e : A$ is derivable.*

Finally, the calculus is consistent. This can be established by a standard model construction, e.g. the proof-irrelevance model of [19].

Proposition 12. *There is no M such that $A : \mathbf{Prop} \vdash M : A$.*

We conjecture, but do not prove, that every legal term is $\beta\beta^+\iota\sigma$-strongly normalising. The most direct way to prove the conjecture seems to adapt the model

constructions of e.g. [24, 39]. An alternative would be to define a reduction-preserving translation from the extended type theory to the original system; however it is unclear to the authors on how to achieve such a result (whose interest goes beyond strong normalisation).

4 Conclusion

We have proposed a computational interpretation of type isomorphisms and detailed a number of situations in which such a mechanism allows proof reuse. For these examples, we have shown, up to the conjecture of strong normalisation, that this mechanism extends safely the Calculus of Inductive Constructions.

Our work suggests a new approach to (some form of) proof reuse and contributes to the understanding of implicit coercions. Yet much work remains to be done:

- from a practical perspective, it seems important to integrate the techniques proposed in this paper into a proof development system such as Coq, and to understand the interactions between our approach and the proof transformation techniques of [30, 33];
- from a theoretical perspective, the outstanding question left unaddressed in this paper is the normalisation of the extended reduction relation. Besides, a number of technical developments seem much desirable. First, we are interested in understanding how the approach developed in this paper might be generalised (1) to a larger class of isomorphisms of types (2) beyond isomorphisms of types. W.r.t. the latter, one may drop the symmetry of isomorphisms of types and consider examples with unidirectional coercions. This would lead us to a calculus where implicit coercions determine a subtyping relation between datatypes and/or record types.
  In a different direction, it is also of interest to spell out the connections between our use of type isomorphisms in the example of vectors and H. Xi and F. Pfenning's use of constraints in DML [40]. This requires to understand under which circumstances the example of vectors, which does not require new computational rules at the term level, can be generalised.

Acknowledgements The authors are grateful to Yves Bertot, Venanzio Capretta, Nicolas Magaud, Femke van Raamsdonk and the anonymous referees for their constructive comments on the paper. Olivier Pons is supported by the Portuguese Science Foundation (Fundação para a Ciencia e a Technologia) under the Fellowship PRAXIS-XXI/BPD/22108/99. The authors also acknowledge support from the cooperation program ICCTI/INRIA.

References

1. J. C. B. Almeida. A formalization of RSA in Coq, July 1999. Available from http://logica.di.uminho.pt/CryptoCoq/index.html.

2. L. Augustsson. Cayenne: A language with dependent types. In *Proceedings of ICFP'98*, pages 239–250. ACM Press, 1998.
3. A. Bailey. *The Machine-Checked Literate Formalisation of Algebra in Type Theory.* PhD thesis, University of Manchester, 1998.
4. B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, P. Loiseleur, C. Muñoz, C. Murthy, C. Parent-Vigouroux, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant User's Guide. Version 6.3.1*, December 1999.
5. G. Barthe. Implicit coercions in type systems. In Berardi and Coppo [10], pages 16–35.
6. G. Barthe. Theoretical pearl: type-checking injective pure type systems. *Journal of Functional Programming*, 9:675–698, 1999.
7. G. Barthe, V. Capretta, and O. Pons. Setoids in type theory. Submitted, 2000.
8. G. Barthe and M.J. Frade. Constructor subtyping. In D. Swiestra, editor, *Proceedings of ESOP'99*, volume 1576 of *Lecture Notes in Computer Science*, pages 109–127. Springer-Verlag, 1999.
9. G. Barthe, M. Ruys, and H. Barendregt. A two-level approach towards lean proof-checking. In Berardi and Coppo [10], pages 16–35.
10. S. Berardi and M. Coppo, editors. *Proceedings of TYPES'95*, volume 1158 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
11. G. Betarte. *Dependent Record Types and Algebraic Structures in Type Theory.* PhD thesis, Department of Computer Science, Chalmers Tekniska Högskola, 1998.
12. F. Blanqui, J.-P. Jouannaud, and M. Okada. The algebraic calculus of constructions. In P. Narendran and M. Rusinowitch, editors, *Proceedings of RTA'99*, volume 1631 of *Lecture Notes in Computer Science*, pages 301–316. Springer-Verlag, 1999.
13. P. Callaghan and Z. Luo. Plastic: An implementation of typed LF with coercive subtyping and universes. *Journal of Automated Reasoning*, 200x. To appear.
14. V. Capretta. Universal algebra in type theory. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Proceedings of TPHOL'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 131–148. Springer-Verlag, 1999.
15. V. Capretta. Recursive families of inductive types. In M. Aagard and J. Harrison, editors, *Proceedings of TPHOLs'00*, volume 1869 of *Lecture Notes in Computer Science*, pages 73–89. Springer-Verlag, 2000.
16. L. Cardelli. Type systems. *ACM Computing Surveys*, 28(1):263–264, March 1996.
17. L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.
18. P. Cohn. *Universal algebra*, volume 6 of *Mathematics and its Applications.* D. Reidel, 1981.
19. T. Coquand. Metamathematical investigations of a calculus of constructions. In P. Odifreddi, editor, *Logic and Computer Science*, pages 91–122. Academic Press, 1990.
20. R. Curien. *Outils pour la preuve par analogie.* PhD thesis, Université de Nancy, 1995.
21. D. Delahaye, R. di Cosmo, and B. Werner. Recherche dans une bibliothèque de preuves Coq en utilisant le type et modulo isomorphismes. In *Proceedings of the PRC/GDR de programmation, Pôle Preuves et Spécifications Algébriques*, November 1997.
22. R. Di Cosmo. *Isomorphisms of types: from λ-calculus to information retrieval and language design.* Progress in Theoretical Computer Science. Birkhauser, 1995.

23. A. Felty and D. Howe. Generalization and reuse of tactic proofs. In F. Pfenning, editor, *Proceedings of LPAR'94*, volume 822 of *Lecture Notes in Artificial Intelligence*, pages 1–15. Springer-Verlag, 1994.
24. H. Geuvers. A short and flexible proof of strong normalisation for the Calculus of Constructions. In P. Dybjer, B. Nordström, and J. Smith, editors, *Proceedings of TYPES'94*, volume 996 of *Lecture Notes in Computer Science*, pages 14–38. Springer-Verlag, 1995.
25. R. W. Hasker and U. S. Reddy. Generalization at higher types. In D. Miller, editor, *Proceedings of the Workshop on the λProlog Programming Language*, pages 257–271. University of Pennsylvania, July 1992. Available as Technical Report MS-CIS-92-86.
26. J.W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: Introduction and survey. *Theoretical Computer Science*, 121(1-2):279–308, 1993.
27. T. Kolbe and C. Walther. Proof analysis, generalization, and reuse. In W. Bibel and P. H. Schmidt, editors, *Automated Deduction: A Basis for Applications. Volume II, Systems and Implementation Techniques.* Kluwer Academic Publishers, 1998.
28. Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9:105–130, February 1999.
29. Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, LFCS, University of Edinburgh, May 1992.
30. N. Magaud and Y. Bertot. Changements de représentation des structures de données dans Coq: le cas des entiers naturels. In P. Casteran, editor, *Proceedings of JFLA'01*, 2001.
31. R. Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994.
32. R. Pollack. Dependently typed records for representing mathematical structures. In M. Aagard and J. Harrison, editors, *Proceedings of TPHOLs'00*, volume 1869 of *Lecture Notes in Computer Science*, pages 462–479. Springer-Verlag, 2000.
33. O. Pons. *Conception et réalisation d'outils d'aide au développement de grosses théories dans les systèmes de preuves interactifs.* PhD thesis, Conservatoire National des Arts et Métiers, 1999.
34. M. Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1):71–89, 1991.
35. E. J. Rollins and J. M. Wing. Specifications as search keys for software libraries. In K. Furukawa, editor, *Proceedings of ICLP'91*, pages 173–187. MIT Press, June 1991.
36. A. Saibi. Typing algorithm in type theory with inheritance. In *Proceedings of POPL'97*, pages 292–301. ACM Press, 1997.
37. P. Severi. Type inference for pure type systems. *Information and Computation*, 143(1):1–23, May 1998.
38. N. Shankar, S. Owre, and J.M. Rushby. *The PVS Proof Checker: A Reference Manual.* Computer Science Laboratory, SRI International, February 1993. Supplemented with the PVS2 Quick Reference Manual, 1997.
39. B. Werner. *Méta-théorie du Calcul des Constructions Inductives.* PhD thesis, Université Paris 7, 1994.
40. H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of POPL'99*, pages 214–227. ACM Press, 1999.