To appear in Math. Struct. in Comp. Science

A short survey of Isomorphisms of Types

Roberto DI COSMO

Preuves Programmes Systèmes
CNRS – Université Paris VII
UMR 7126 – Case 7014
2, place Jussieu – 75251 Paris Cedex 05 – FRANCE
http://www.dicosmo.org

Received 12 April 2005

Contents

1. Introduction

We have all be taught in high school that two objects A and B are *isomorphic* iff there exists two functions f and g such that

$$\begin{tikzcd} A \arrow[loop left, "id_A"] \arrow[r, bend left, "f"] & B \arrow[loop right, "id_B"] \arrow[l, bend left, "g"] \end{tikzcd}$$

Here we are particularly interested in *type isomorphisms*, which arise when A and B are *types* of some (abstract) programming language, like the typed λ-calculus, even if, by the well known Curry-Howard correspondence, these types can also be seen as *formulae* of some logic, or even *objects* in some category, so that looking for isomorphisms in either one of these fields will bring results in all the others. As a simple example of this phenomenon, consider the case of the arrow and product type constructors, and their equivalent in logic and category theory, as summarised in the following table:

Type	*Proposition*	*Categorical object*
$A \rightarrow B$	$A \supset B$	B^A
$A \times B$	$A \wedge B$	$A \times B$

then, the currying isomorphism $(A \times B) \rightarrow C = A \rightarrow (B \rightarrow C)$, well known to all functional programmers, becomes the isomorphism of objects $C^{(A\times B)} = (C^B)^A$, well known to all category theorists, and the strong equivalence of propositions $(A \wedge B) \supset C = A \supset (B \supset C)$, well known to all proof theorists.

Building models satisfying specific isomorphisms of types (or domain equations) was a crucial problem in the denotational semantics of programming languages, but in the 1980s some interest started to develop around the dual problem of finding the domain equations (type isomorphisms) that must hold in *every* model of a given language, or *valid isomorphisms of types*, as they were called in (BL85).

There are essentially two families of techniques to address this question: one can work *syntactically* to characterize those programs f that possess an inverse g making the above diagram commute, or one can work *semantically* trying to find some specific model that captures the isomorphisms valid in all models. The next two sections of this survey are dedicated to these two approaches.

But the study of type isomorphisms is now a well established research field with many ramifications, as the variety of the subjects tackled by the papers selected for this journal issue clearly shows, so we will also give space to applications, complexity results, and open problems.

2. Type isomorphisms and invertible terms

If we want to pinpoint some early dates in the long story of this study, it is natural to start with Dezani's seminal work (Dez76), back in 1976, on the untyped lambda calculus. Her deep, technical syntactical analysis characterized fully the *invertible terms*, the terms M for which a term M^{-1} exists s.t. $\lambda x.M^{-1}(Mx) =_{\beta\eta} \lambda x.M(M^{-1}x) =_{\beta\eta} \lambda x.x$, as the *finite hereditary permutations*, a class of terms which can be easily defined inductively, and that can be seen as a family of generalized η-expansions.

While this work was done in the framework of the untyped lambda calculus, it turned out that this family of invertible terms *can be typed* in the simple typed lambda calculus, and this allowed Bruce and Longo (BL85) to prove by a straightforward induction on the structure of the finite hereditary permutations that in the simple typed lambda calculus the only type isomorphisms w.r.t. $\beta\eta$ equality are those induced by the *swap* equation $A \to (B \to C) = B \to (A \to C)$.

By extending Dezani's original technique to the invertible terms in typed calculi with additional constructors (like products and unit type) or higher order (System F or Core-ML), it has been possible to pursue this line of research to the point of getting a full characterization of isomorphisms in a whole set of typed lambda calculi, from $\lambda^1\beta\eta$, that correspond to $IPC(\Rightarrow)$, the intuitionistic positive calculus with implication, whose isomorphisms are described by Th^1 (Mar72; BL85), to $\lambda^1\beta\eta\pi*$, that corresponds to Cartesian Closed Categories and $IPC(\mathbf{True}, \wedge, \Rightarrow)$, for which $Th^1_{\times T}$ is complete (BDCL90)†, to $\lambda^2\beta\eta$ (System F), that corresponds to $IPC(\forall, \Rightarrow)$, and whose isomorphisms are given by Th^2 (BL85), to $\lambda^2\beta\eta\pi*$ (System F with products and unit type), that corresponds to $IPC(\forall, \mathbf{True}, \wedge, \Rightarrow)$, whose isomorphisms are given by $Th^2_{\times T}$ (DC91). A summary of the axioms in these theories is given in table 1.

The focus, in this line of research, is to find all the type isomorphisms for a given language (λ-calculus) and a given notion of equality on terms (which almost always contains extensional rules like η, as otherwise no nontrivial invertible term exists (Dez76)) as a consequence of an inductive characterization of the invertible terms.

Notice that the type isomorphisms which correspond to invertible terms (called *definable isomorphisms of types in (BL85)*) are *a priori* not the same as the *valid isomorphisms of types*: a definable isomorphism seems a stronger notion, demanding that a given isomorphism not only hold in all models, but that it also holds *uniformly* in all models.

Nevertheless, in all the cases studied in the literature, it is easy to build a free model out of the calculus, and to prove that valid and definable isomorphisms coincide, so this distinction has gradually disappeared in time.

One notable missing piece in the table summarizing the theory of isomorphisms of types is the case of intersection types: we know already the form of the invertible terms, as they are again Dezani's finite hereditary permutations, yet the intersection type discipline can give many widely different typings for the same term, so that the simple proof technique in (BL85) does not apply, and a complete theory of isomorphisms for them is not yet known.

3. From Tarski's High School Algebra problem to isomorphisms in category theory

Another line of research that led to fundamental results in the field of isomorphisms of types can be traced back to Soloviev's seminal work (Sol83) on isomorphic objects in Cartesian Closed Categories, where he proves that such isomorphisms are exactly the ones generated by the theory $Th^1_{\times T}$. To prove this result, Soloviev first notices that the

† But this result had been proved earlier by Soloviev using model theoretic techniques, see next section.

$$\left.(\mathbf{swap})\quad A \to (B \to C) = B \to (A \to C)\right\} Th^1$$

$$\left.\begin{array}{l}
\left.\begin{array}{l}
\left.\begin{array}{ll}
1. & A \times B = B \times A \\
2. & A \times (B \times C) = (A \times B) \times C \\
3. & (A \times B) \to C = A \to (B \to C) \\
4. & A \to (B \times C) = (A \to B) \times (A \to C) \\
5. & A \times \mathbf{T} = A \\
6. & A \to \mathbf{T} = \mathbf{T} \\
7. & \mathbf{T} \to A = A
\end{array}\right\} Th^1_{\times T} \\
\left.\begin{array}{ll}
8. & \forall X.\forall Y.A = \forall Y.\forall X.A \\
9. & \forall X.A = \forall Y.A[Y/X] \\
10. & \forall X.(A \to B) = A \to \forall X.B
\end{array}\right\} + \mathbf{swap} \ = Th^2 \\
\begin{array}{ll}
11. & \forall X.A \times B = \forall X.A \times \forall X.B \\
12. & \forall X.\mathbf{T} = \mathbf{T}
\end{array}
\end{array}\right\} Th^2_{\times T} \\
\mathbf{split} \quad \forall X.A \times B = \forall X.\forall Y.A \times (B[Y/X])
\end{array}\right\} - 10, 11 = Th^{ML}$$

Table 1. *Type isomorphisms in typed lambda calculi*
N.B.: in equation 8, X must be free for Y in A and $Y \notin FTV(A)$; in equation 10, $X \notin FTV(A)$.

isomorphisms in $Th^1_{\times T}$ hold in any cartesian closed category, and then pinpoints one particular Cartesian Closed Category, the Category of Finite Sets, where only the isomorphisms of $Th^1_{\times T}$ hold, thus concluding the proof[‡].
Rittri and others later pointed out that Soloviev's work was related to the well known Tarski's High School Algebra Problem, where one is concerned with finding all the valid equalities over the natural numbers that can be described using a given language (with or without product, exponentiation, sums, constants for one or zero, etc.). Indeed, in the Category of Finite Sets, objects are sets, which are isomorphic only if they have the same cardinality, and when seen as a cartesian closed category, these isomorphisms exactly correspond to equations on the cardinalities written using a constant for the integer one, multiplication and exponentiation.

Later on, Soloviev (Sol93) gave a complete axiomatization of isomorphisms in Symmetric Monoidal Closed Categories, using proof theoretic techniques, and Dosen and Petric (DP97) provided an arithmetical structure that exactly corresponds to these isomorphisms.

3.1. *Tarski's High School Algebra Problem*

In 1969, Tarski (DT69) asked if the equational theory $\mathcal{E}$ of the usual arithmetic identities of figure 1 that are taught in high school are complete for the standard model $\langle \mathbb{N}, 1, +, \times, \uparrow \rangle$ of positive natural numbers; i.e. , if they are enough to prove all the arithmetic identities (he considered zero fundamental too, but, probably due to the presence of one conditional

[‡] See also (MS90) for a different proof.

equation, he left for further investigation the case of the other equations of figure 2, that we are also taught in high school).

$$(\mathcal{E}_1)\quad 1\times x = x \qquad (\mathcal{E}_2)\quad x\times y = y\times x \qquad (\mathcal{E}_3)\quad (x\times y)\times z = x\times(y\times z)$$

$$(\mathcal{E}_4)\quad x^1 = x \qquad (\mathcal{E}_5)\quad 1^x = 1$$
$$(\mathcal{E}_6)\quad x^{y\times z} = (x^y)^z \qquad (\mathcal{E}_7)\quad (x\times y)^z = x^z\times y^z$$

$$(\mathcal{E}_8)\quad x+y = y+x \qquad (\mathcal{E}_9)\quad (x+y)+z = x+(y+z)$$
$$(\mathcal{E}_{10})\quad x\times(y+z) = x\times y + x\times z \qquad (\mathcal{E}_{11})\quad x^{(y+z)} = x^y\times x^z$$

Fig. 1. Equations without zero

$$(\mathcal{Z}_1)\quad 0\times x = 0 \qquad (\mathcal{Z}_2)\quad 0+x = x \qquad (\mathcal{Z}_3)\quad x^0 = 1$$

$$(\mathcal{Z}_4)\quad 0^x = 0 \quad (x>0)$$

Fig. 2. Equations and conditional equation for zero

He conjectured that they were[§], but was not able to prove the result. Martin (Mar72) showed that the identity $(\mathcal{E}_6)$ is complete for the standard model $\langle \mathbb{N}, \uparrow\rangle$ of positive natural numbers with exponentiation, and that the identities $(\mathcal{E}_2)$, $(\mathcal{E}_3)$, $(\mathcal{E}_6)$, and $(\mathcal{E}_7)$ are complete for the standard model $\langle \mathbb{N}, \times, \uparrow\rangle$ of positive natural numbers with multiplication and exponentiation. He also exhibited the identity

$$(x^u + x^u)^v \times (y^v + y^v)^u = (x^v + x^v)^u \times (y^u + y^u)^v$$

that in the language without the constant 1 is not provable in $\mathcal{E}$. [¶] The question was not completely settled by this counterexample, because it is was only a counterexample in the language without a constant for 1, that Tarski clearly considered necessary in his paper, as well as the constant for 0, even if he did not explicitly mention it in his conjecture. In the presence of a constant 1, the following new equations come into play, and allow us to easily prove Martin's equality.

$$1a = a \quad a^1 = a \quad 1^a = 1$$

This problem attracted the interest of many other mathematicians, like Leon Henkin, who focused on the equalities valid in $\langle \mathbb{N}, 0, +\rangle$, and showed the completeness of the usual known axioms (commutativity, associativity of the sum and the zero axiom), and gives a very nice presentation of the topic in (Hen77).

[§] Actually, he conjectured something stronger, namely that $\mathcal{E}$ is complete for $\langle \mathbb{N}, Ack(n, \_, \_)\rangle$, the natural numbers equipped with a family of generalized binary operators $Ack(n, \_, \_)$ that extend the usual sum $+$, product $\times$ and exponentiation $\uparrow$ operators. In Tarski's definition, $Ack(0, \_, \_)$ is the sum, $Ack(1, \_, \_)$ is multiplication, $Ack(2, \_, \_)$ is exponentiation.

[¶] He also showed that there are no nontrivial equations for $\langle \mathbb{N}, Ack(n, \_, \_)\rangle$ if $n > 2$.

Wilkie (Wil81) was the first to establish Tarski's conjecture in the negative. Indeed, by a proof-theoretic analysis, he showed that the identity

$$(A^x + B^x)^y \times (C^y + D^y)^x = (A^y + B^y)^x \times (C^x + D^x)^y$$

where $A = 1 + x$, $B = 1 + x + x^2$, $C = 1 + x^3$, $D = 1 + x^2 + x^4$ is not provable in $\mathcal{E}$.

R. Gurevič later gave an argument by an ad hoc counter-model (Gur85) and, more importantly, showed that there is no finite axiomatization for the valid equations in the standard model $\langle \mathbb{N}, 1, +, \times, \uparrow \rangle$ of positive natural numbers with one, multiplication, exponentiation, and addition (Gur90). He did this by producing an infinite family of equations such that for every sound finite set of axioms one of the equations can be shown not to follow. Gurevič's identities, which generalize Wilkie's identities, are the following

$$\left(A^x + {B_n}^x\right)^{2^x} \times \left({C_n}^{2^x} + {D_n}^{2^x}\right)^x \quad = \quad \left(A^{2^x} + {B_n}^{2^x}\right)^x \times \left({C_n}^x + {D_n}^x\right)^{2^x}$$

where

$$\begin{array}{rcl} A & = & 1 + x \\ B_n & = & 1 + x + \cdots + x^{n-1} \; = \; \sum_{i=0}^{n-1} \; x^i \\ C_n & = & 1 + x^n \\ D_n & = & 1 + x^2 + \cdots + x^{2(n-1)} \; = \; \sum_{i=0}^{n-1} \; x^{2i} \\ n & \geq & 3 \quad \text{is odd} \end{array}$$

Nonetheless, equality in all these structures, even if not finitely axiomatizable, was shown to be decidable (Mac81; Gur85), while in (HR84) one can find a subclass of numerical expressions for which the usual axioms for $+$, $\times$, $\uparrow$ and 1 are complete.

Balat, Fiore and this author investigated the question as to whether the correspondence between numerical equalities and type isomorphisms was limited to the case of the well-behaved unit, product, and arrow type constructors and, in particular, if it could be extended to the more problematic types involving the empty type and the sum type constructor (BDCF02; BDCF04), with the following fundamental result:

> Gurevič's identities are indeed type isomorphisms, and one can then show that the theory of type isomorphisms in the presence of the product, arrow, and sum type constructors, and the unit type is not finitely axiomatizable.

This result has been pursued further by Fiore (Fio04), who is now studying the connections with *objective number theory* as advocated by Schanuel and Lawvere.

Finally, Dufour and this author show in (DCD05) that $\langle \mathbb{N}, 0, 1, +, \times, \uparrow \rangle$ has a decidable, but not finitely axiomatizable, equational theory, and that the only difference between $\langle \mathbb{N}, 0, 1, +, \times, \uparrow \rangle$ and $\langle \mathbb{N}, 1, +, \times, \uparrow \rangle$ is given by the traditional equations and conditional equation for zero.
As a consequence, the family of Gurevič's equalities does not collapse, and we also know now that the theory of type isomorphisms in Bi-Cartesian Closed Categories is not finitely axiomatizable. We do not know if this theory, like for the integers, is decidable.

3.2. *Complexity and decidability issues*

The theories of type isomorphisms in Table 1 are all decidable, but what about their complexity? Dropping the nonlinear axioms in $Th^1_{\times T}$, Soloviev first showed (SA94; SA97) that one can get an efficient decision procedure running in $O(n\log^2(n))$ time. Due to the nonlinear axioms involving the product type, one could expect a much higher complexity for the full $Th^1_{\times T}$; this is not the case, as Zibin, Gil and Considine provide in the paper included in this issue a very efficient $O(n\log n)$ decision procedures for this system.
Nevertheless, if we are interested in matching, or unification, up to these theories, the complexity goes up: matching up to $Th^1_{\times T}$ is decidable, as shown by Rittri (Rit90) using the techniques originally developed in a technical report by Bernard Lang that we publish here for the first time, while unification is undecidable, matching and linear unification are NP-Complete (NPS93; NPS97).

4. Isomorphisms in logic

If we look through the Curry-Howard correspondence mirror, we can rephrase the type isomorphism problem in proof-theoretical terms. Now, two propositions A and B are *isomorphic* if there exist two proofs π_A of $B \vdash A$ and π_B of $A \vdash B$ such that by cutting π_A and π_B on the conclusion B (resp. A) we obtain a proof equal, up to some fixed equality of proofs, to the axiom $A \vdash A$ (resp. $B \vdash B$).

Of course, isomorphic propositions are logically equivalent, but the converse is not true, and this is why the term *strong equivalence* has been used in the literature instead (Mar92; BM94).

Since the typed lambda calculi examined in the first section correspond to the (variants of) intuitionistic positive propositional calculus, all the results on type isomorphisms recalled there immediately translate to the corresponding intuitionistic calculus, without further ado.

But the situation is different for logics, like Linear Logic (Gir87), for which the λ-calculus is not the natural corresponding computational paradigm. There, one needs to redo the work from scratch, and it is possible to characterize the *invertible* proof nets and show that for MLL, the multiplicative fragment of Linear Logic, the following theory is complete:

$$\begin{array}{rcl@{\qquad\qquad}rcl} X\otimes Y & = & Y\otimes X & (X⅋Y)⅋Z & = & X⅋(Y⅋Z) \\ X⅋Y & = & Y⅋X & (X\otimes Y)\otimes Z & = & X\otimes(Y\otimes Z) \\ X\otimes 1 & = & X & X⅋\bot & = & X \end{array}$$

This provides a nice symetrization of the isomorphisms for Cartesian Closed Categories: when reading $A \to B = A^\perp ⅋ B$, $A \times B = A\otimes B$ and $unit = 1$, the theory $Th^1_{\times T}$ reduces, through Linear Logic's looking mirror, to just the associativity, commutativity and unit rules of MLL. It is an open problem to extend this simple characterization to larger fragments of Linear Logic.

If one looks at Polarised Linear Logic (LLP), a finite characterization of the strong equivalence of propositions for LLP can be obtained by means of a very elegant usage of game semantics, a result presented in Laurent's paper included in this issue, that provides a nice example of a proof along the lines of the *semantic* tradition as opposed to the proof for MLL. Laurent also points to a possible interpretation *à la Tarski* of these isomorphisms into the real numbers.

Building on the result for LLP, Laurent also provides a finite, complete characterization of classical isomorphic propositions, including disjunction.

This is not contradictory with the non finite axiomatizability of type isomorphisms with the sum type, as the classical disjunction and the sum type are not the same operator; in category theoretical terms, on one side we work in Bi-CCCs, on the other we find Control Categories. Laurent's result is extended to second order classical logic by de Lataillade (dL04).

Finally, some recent work has begun to explore the possibilities offered by a mixed approach, where one adds to a lambda calculus with inductive types new reductions to realize some chosen isomorphism, as in the work by Chemouil that appears in this issue, that follows the lines of (SC03).

5. Practical applications

Isomorphisms of types have several interesting practical applications, ranging from library search, to correcting type errors.

5.1. *Types as search keys*

Rittri was the first to propose to use isomorphisms of types as a key tool to retrieve library components. He pointed out that a function in a library can have a type syntactically quite different from the one expected by the user, as shown in his famous example for the functional list iterator:

Language	Name	Type
ML of Edinburgh LCF	itlist	$('a \to 'b \to 'b) \to 'a\ list \to 'b \to 'b$
CAML	list_it	"
Haskell	foldl	$('a \to 'b \to 'a) \to 'a \to 'b\ list \to 'a$
Ocaml	List.fold_left	"
SML of New Jersey	fold	$('a \times 'b \to 'b) \to 'a\ list \to 'b \to 'b$
Edinburgh SML Library	fold_left	$('a \times 'b \to 'b) \to 'b \to 'a\ list \to 'b$

All the types in this table are different, *but isomorphic*, so he proposed to search functions using their type modulo isomorphism as a key (Rit91), and he explored the possibilities offered by matching and linear unification of types modulo type isomorphisms using the theory $Th^1_{\times T}$ (Rit92; Rit93). This author implemented a search tool along the same lines, but using Th^{ML}, in the Caml programming language, while a similar tool has been built for CamlLight by Jerome Vouillon: using the command line tool `camlsearch`, one can query the standard library to find the list iterator as in the following example, where the system retrieves for us two good candidates for a list iterator, `list_it` and `it_list`:

```
camlsearch -s -e "'b*'a list*('b->'a->'b) -> 'b"

  it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

  list_it : ('b -> 'a -> 'a) -> 'b list -> 'a -> 'a
```

Following this line of applications, one finds also a study of search tools for the library of theorems of the Coq proof assistant in (DDCW97; Del99; BP01b) and in Barthe's paper in this journal issue.

5.2. *Building coercions*

Finding the library object satisfying a query up to isomorphisms is not the full story: in order to use it in the context of her program, the user must build some *glue code* which is exactly the pair of invertible terms (the coercions) that realize the isomorphism. While for simple functions this is a reasonably straightforward business, when one turns to more sophisticated languages, or language constructs, like classes, objects and modules, or dependent types in proof assistants (BP01a), building the coercions can become a taunting task.

This is why a whole line of articles have been dedicated to automatically building coercions (AJ04) for type systems which are quite sophisticated.

For a language with modules and functors (Mul; ZW93), isomorphisms can get very intricate (ADC96; ADCD97), as they can come from the base language, the module language, or both, as shown by the following isomorphic functor signatures

```
module UnifyCurry :
   functor (t:TERMS) ->
      functor (s:SUBSTITUTION with s.termtype = t.termtype):
      sig
           val c: int*string
          unify: t.termtype -> t.termtype -> s.substtype
      end

module UnifyUnCurry :
   functor (sig module t:TERMS module s:SUBSTITUTION
                with s.termtype = t.termtype end):
   sig
        val c1: int
        val c2: string
        findunifier: t.termtype * t.termtype -> s.substtype
   end
```

An attempt at building an efficient search tool taking these issues into account has been done by Yakobowski (Yak02).

One can also build glue code using type isomorphisms as an alternative to Interface Definition Languages, as done in IBM's Mockingbird project (ACC97; ABR98) where sums and recursive types are needed to provide a type system powerful enough to represent all the type constructs in the source programming languages (like C or Java).

As for classes, it is possible to provide useful tools even when restricting attention to very basic isomorphisms, like associativity and commutativity of product as done in (Tha94; PZ00; JPZ02; DCPR05), that allow to avoid tackling the very complex issue of isomorphisms of recursive types directly.

In all these more sophisticated applications, no claim of completeness is made: finding all type isomorphisms in a language that allows recursion, sum types and/or subtyping might well be undecidable, unless we impose some restrictions on the expressiveness of the coercions; for example, one could restrict these coercions to only use iterators on well founded recursive types, and not full recursion, along the lines of (Fio04), which seems the most promising approach to date.

5.3. *Type isomorphisms inside the type system*

Finally, another line of research has concentrated on using type isomorphisms directly *inside* a typed programming language, as opposed to the usage of some *external* tool. One line of research uses isomorphisms to perform transformations of data types inside the language: the earliest proposals in this direction is surely Wadler's seminal paper on views (Wad87), where isomorphisms are used as a tool to provide a correspondence between an externally available concrete representation of an abstract data type, over which the programmer can use pattern matching, and the internal, hidden implementation; more recently, it has been proposed to use isomorphisms to allow transformations over data types in XML documents (AJ).

Along a different line, one can use isomorphisms to change the type system, either by directly incorporating them in the type system as in work by Nipkow (Nip90), or by using type isomorphisms to automatically correct typing errors, as originally proposed in (Cos86) where a linear time algorithm was used to correct errors when the coercion is unique, and more recently done by MacAdam (McA02).

6. Recursive types

The Mockingbird project acted as a very powerful motivation to investigate isomorphisms in the presence of recursive types, which forced researchers to understand better what goes on when the implementation of a library search tool decides that isomorphisms simply percolate through user defined types, as is the case when returning a value of type $(A \times B)$ *list* when presented the query $(B \times A)$ *list*. Intuitively, we know that it will suffice to *map* the coercions for the commutativity of product across the list, but it is not evident that such a *map* function will exist in general for all user defined types, even when not considering the complication introduced by restrictions of visibility like those introduced by abstract data types. What we really want is to be able to perform a derivation like the following one, where the middle equality is an isomorphism of recursive types

$$\begin{aligned} A \times B\, list &= \mu X.(A \times B) \times X + 1 \\ &= \mu X.(B \times A) \times X + 1 = B \times A\, list \end{aligned}$$

But isomorphisms of recursive types are quite tricky, as they come in different kinds:

identity isomorphisms capture the equivalence among different syntactic representations of the same object, like in

$$\mu X.A \times X = \mu X.A \times (A \times X)$$

where the two sides are interpreted by the same infinite tree. For identity isomorphisms of recursive types (AC93; AF96) propose a complete deductive system that includes the rule:

$$\frac{A = F(A)}{A = \mu X.F(X)} \qquad (fold)$$

isomorphisms realized by the identity capture the equivalence of types A and B which are not interpreted in the model by the same object, but whose coercions have no computational content, like for the isomorphism

$$\forall X.\forall Y.A = \forall Y.\forall X.A$$

These coercions without computational content typically arise in the presence of polymorphism and are known as *retyping functions* (Mit88).

proper isomorphisms are those where the coercions have computational content, like for

$$A \times B = B \times A$$

These different kinds of isomorphisms must be distinguished with care to avoid inconsistencies. If one mix for example the *fold* rule for identity isomorphisms with the proper isomorphisms for the terminal object $A = A \times 1$, it is easy to infer that all types are isomorphic to $\mu X.X \times 1$, which is clearly false.

To validate the transformations performed by the Mockingbird system, it is enough to use a sequence of equality steps, each of which can be either an identity isomorphism or a proper one, and the induced equality is consistent (as proved by P.M. Lopez).

Nevertheless, it would be quite important to establish a consistent formal reasoning system able to tackle the full power of type isomorphisms in the presence of recursive types, and possibly to allow to prove conditional isomorphisms like $(A \times B)$ *list* $=$ $(A$ *list*$) \times (B$ *list*$)$ when the two lists have the same length; in this direction, there is clearly some connection to be established with work related to polytypism and generic programming like in (JJ96; JJ02; MBJ99).

7. Conclusions

We have shown in this short survey that the study of isomorphisms of types is a very lively research subject, that has attracted the interest of researchers coming from very diverse fields, from logic to lambda calculus, to algorithmics and programming.
Isomorphisms of types have concrete applications in the field of programming languages and type systems, that have raised new open problems that we believe will feed the research over the next years.

References

Joshua Auerbach, Charles Barton, and Mukund Raghavachari. Type isomorphisms with recursive types. Technical Report RC 21247, IBM Yorktown Heights, 1998.

Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

Joshua Auerbach and Mark C. Chu-Carrol. The mockingbird system: a compiler-based approach to maximally interoperable distributed systems. Technical Report RC 20718, IBM Yorktown Heights, 1997.

Maria-Virginia Aponte and Roberto Di Cosmo. Type isomorphisms for module signatures. In *Programming Languages Implementation and Logic Programming (PLILP)*, volume 1140 of *Lecture Notes in Computer Science*, pages 334–346. Springer-Verlag, 1996.

Maria-Virginia Aponte, Roberto Di Cosmo, and Catherine Dubois. Signature subtyping modulo type isomorphisms. submitted, 1997.

Martìn Abadi and Marcelo P. Fiore. Syntactic considerations on recursive types. In *Proceedings of the Symposium on Logic in Computer Science (LICS)*, number 11, pages 242–252, New Brunswick, New Jersey, 1996. IEEE computer society Press.

Frank Atanassow and Johan Jeuring. Type isomorphisms simplify xml programming. Available as `http://www.win.tue.nl/ipa/archive/falldays2003/planx04.pdf`.

Frank Atanassow and Johan Jeuring. Inferring type isomorphisms generically. In Dexter Kozen, editor, *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 2004, Proceedings*, number 3125 in LNCS, pages 32–53, Berlin Heidelberg, July 2004. Springer–Verlag.

Vincent Balat, Roberto Di Cosmo, and Marcelo Fiore. Remarks on isomorphisms in typed lambda calculi with empty and sum type. In *LICS*. IEEE, July 2002.

Vincent Balat, Roberto Di Cosmo, and Marcelo Fiore. Extensional normalisation and type-directed partial evaluation for typed lamda calculus with sums. In *31st Ann. ACM Symp. on Principles of Programming Languages (POPL)*, pages 64–76. ACM, 2004.

Kim Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. Technical Report 90-14, LIENS - Ecole Normale Supérieure, 1990.

Kim Bruce and Giuseppe Longo. Provable isomorphisms and domain equations in models of typed languages. *ACM Symposium on Theory of Computing (STOC 85)*, 1985.

Franco Barbanera and Simone Martini. Proof-functional connectives and realizability. *Archive of Mathematical Logic*, 33:189, 1994.

Gilles Barthe and Olivier Pons. Type isomorphisms and proof reuse in dependent type theory. In *FOSSACS*, 2001.

Gilles Barthes and Olivier Pons. Type isomorphisms and proof reuse in dependent type theory. In Furio Honsell and M Miculan, editors, *FOSSACS*, number 2030 in LNCS, pages 57–71. Springer-Verlag, 2001.

Roberto Di Cosmo. Isomorfismi di tipi. Master's thesis, Università di Pisa, 1986.

Roberto Di Cosmo. Invertibility of terms and valid isomorphisms. a proof theoretic study on second order λ-calculus with surjective pairing and terminal object. Technical Report 91-10, LIENS - Ecole Normale Supérieure, 1991.

Roberto Di Cosmo and Thomas Dufour. The equational theory of $< n, 0, 1, +, \cdot, \uparrow>$ is decidable, but not finitely axiomatisable. In *LPAR'05*, Lecture Notes in Computer Science, page 240, 2005.

Roberto Di Cosmo, François Pottier, and Didier Rémy. Subtyping recursive types modulo associative commutative products. *Typed Lambda Calculus and Applications*, 2005.

David Delahaye, Roberto Di Cosmo, and Benjamin Werner. Recherche dans une bibliothèque de preuves Coq en utilisant le type et modulo isomorphismes. In *PRC/GDR de programmation, Pôle Preuves et Spécifications Algébriques*, 1997.

David Delahaye. Information Retrieval in a Coq Proof Library using Type Isomorphisms. In *Proceedings of TYPES'99, Lkeberg (Sweden)*, volume 1956 of *LNCS*, pages 131–147. Springer-Verlag, June 1999.

Mariangiola Dezani-Ciancaglini. Characterization of normal forms possessing an inverse in the $\lambda\beta\eta$ calculus. *Theoretical Computer Science*, 2:323–337, 1976.

Joachim de Lataillade. Isomorphismes de second ordre dans les jeux. Master's thesis, DEA Programmation, 2004. Submitted to CSL 2005.

Kosta Dosen and Zoran Petric. Isomorphic objects in symmetric monoidal closed categories. *Mathematical Structures in Computer Science*, 7(6):639–662, 1997.

John Doner and Alfred Tarski. An extended arithmetic of ordinal numbers. *Fundamenta Mathematica*, 65:95–127, 1969.

Marcelo P. Fiore. Isomorphisms of generic recursive polynomial types. In Neil D. Jones and Xavier Leroy, editors, *POPL*, pages 77–88. ACM, 2004.

Joseph (Yossi) Gil. Subtyping arithmetical types. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 276–289, New York, NY, USA, 2001. ACM Press.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50, 1987.

R. Gurevič. Equational theory of positive numbers with exponentiation. *Proceedings of the American Mathematical Society*, 94(1):135–141, 1985.

R. Gurevič. Equational theory of positive numbers with exponentiation is not finitely axiomatizable. *Annals of Pure and Applied Logic*, 49:1–30, 1990.

Leon Henkin. The logic of equality. *American Mathematical Monthly*, 84:597–612, October 1977.

C. W. Henson and L. A. Rubel. Some applications of Nevanlinna theory to mathematical logic: Identities of exponential functions. *Trans. Am. Math. Soc.*, 282(1):1–32, March 1984.

Johan Jeuring and Patrik Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Tutorial Text 2nd Int. School on Advanced Functional Programming, Olympia, WA, USA, 26–30 Aug 1996*, volume 1129, pages 68–114. Springer-Verlag, Berlin, 1996.

Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Sci. Comput. Program.*, 43(1):35–75, 2002.

Somesh Jha, Jens Palsberg, and Tian Zhao. Efficient type matching. In *International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, volume 2303 of *Lecture Notes in Computer Science*, pages 187–204. Springer Verlag, April 2002.

Angus Macintyre. The laws of exponentiation. In C. Berline, K. McAloon, and J.-P. Ressayre, editors, *Model Theory and Arithmetic*, volume 890 of *Lecture Notes in Mathematics*, pages 185–197. Springer-Verlag, 1981.

Charles F. Martin. Axiomatic bases for equational theories of natural numbers. *Notices of the Am. Math. Soc.*, 19(7):778, 1972.

Simone Martini. Provable isomorphisms, strong equivalence and realizability. In Marchetti-Spaccamela et al., editor, *Proceedings of the Fourth Italian Conference on Theoretical Computer Science*, pages 258–268. Word Scientific Publishing Co, 1992.

Eugenio Moggi, Gianna Bellè, and Colin Barry Jay. Monads, shapely functors and traversals. In M. Hoffman, Pavlović, and P. Rosolini, editors, *Proceedings of the Eighth Conference on Category Theory and Computer Science (CTCS'99)*, volume 24 of *Electronic Lecture Notes in Computer Science*, pages 265–286. Elsevier, 1999.

Bruce McAdam. How to repair type errors automatically. In *Trends in functional programming*, pages 87–98. Intellect Books, Exeter, UK, UK, 2002.

John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76:211–249, 1988.

Lambert Meertens and Arno Siebes. Universal type isomorphisms in cartesian closed categories. Centrum voor Wiskunde en Informatica, Amsterdam, the Netherlands. E-mail: lambert,arno@cwi.nl, 1990.

Roberto Muller. A software library architecture based on ml module signatures. Available as `ftp://ftp.cs.bc.edu/pub/muller/library.ps`.

Tobias Nipkow. A critical pair lemma for higher-order rewrite systems and its application to λ^*. *First Annual Workshop on Logical Frameworks*, 1990.

Paliath Narendran, Frank Pfenning, and Rick Statman. On the unification problem for cartesian closed categories. In *Proceedings of the Symposium on Logic in Computer Science (LICS)*, number 8, pages 57–63, Montreal, Canada, 1993. IEEE computer society Press.

Paliath Narendran, Frank Pfenning, and Rick Statman. On the unification problem for Cartesian closed categories. *Journal of Symbolic Logic*, 62:636–647, 1997.

Jens Palsberg and Tian Zhao. Efficient and flexible matching of recursive types. In *Logic in Computer Science*, pages 388–398, 2000.

Mikael Rittri. Retrieving library identifiers by equational matching of types. In *10^{th} Int. Conf. on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

Mikael Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1):71–89, 1991.

Mikael Rittri. Retrieving library functions by unifying types modulo linear isomorphisms. Technical Report 66, Chalmers University of Technology and University of Göteborg, 1992. Programming Methodology Group.

Mikael Rittri. Retrieving library functions by unifying types modulo linear isomorphism. *RAIRO Theoretical Informatics and Applications*, 27(6):523–540, 1993.

Sergei V. Soloviev and Alexander E. Andreev. Linear isomorphism of types: a low upper bound for complexity. Technical report, BRICS reports in Computer Science, 1994.

Sergei V. Soloviev and Alexander E. Andreev. A decision algorithm for linear isomorphism of types with complexity $o(nlog^2(n))$. In *Category Theory and Computer Science*, volume 1290 of *Lecture Notes in Computer Science*, 1997.

Sergei Soloviev and David Chemouil. Some algebraic structures in lambda-calculus with inductive types. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 338–354. Springer, 2003.

Sergei V. Soloviev. The category of finite sets and cartesian closed categories. *Journal of Soviet Mathematics*, 22(3):1387–1400, 1983. English translation of the original paper in russian published in Zapiski Nauchyn Seminarov LOMI, v.105, 1981.

Sergei V. Soloviev. A complete axiom system for isomorphism of types in closed categories. In A. Voronkov, editor, *Logic Programming and Automated Reasoning, 4th International Conference*, volume 698 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 360–371, St. Petersburg, Russia, 1993. Springer-Verlag.

Satish R. Thatté. Automated synthesis of interface adapters for reusable classes. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 174–187, January 1994.

Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In Steve Munchnik, editor, *Proceedings, 14th Symposium on Principles of Programming Languages*, pages 307–312. Association for Computing Machinery, 1987.

Alex J. Wilkie. On exponentiation — A solution to Tarski's high school algebra problem. Math. Inst. Oxford University (preprint), 1981.

Boris Yakobowski. Matching de modules modulo isomorphismes. *Proceedings of the first Workshop on Isomorphisms of Types*, 2002. Available as `http://www.irit.fr/zeno/WIT2002/yakobowski.ps`.

Amy Moormann Zaremsky and Jeannette M. Wing. Signature matching: a key to reuse. In *SIGSOFT*, December 1993. Also available as CMU-CS-93-151, May 1993.