Journal of Functional Programming (1992), *vol.* 11, *pp.* 1–000

Deciding Type Isomorphisms in a type-assignment framework§

ROBERTO DI COSMO
E-mail: roberto@dicosmo.org
LIENS (CNRS) - DMI, Ecole Normale Supérieure, 45 Rue d'Ulm, 75005 Paris - France
and
Dipartimento di Informatica - Corso Italia 40, 56100 Pisa -Italy

Abstract

This paper provides a formal treatment of isomorphic types for languages equipped with an ML style polymorphic type inference mechanism. The results obtained make less justified the commonplace feeling that (the core of) ML is a subset of second order λ-calculus: we can provide an isomorphism of types that holds in the core ML language, but not in second order λ-calculus. This new isomorphism allows to provide a complete (and decidable) axiomatisation of all the types isomorphic in ML style languages, a relevant issue for the *type as specifications* paradigm in library searches. This work is a very extended version of (DC92): we provide both a thorough theoretical treatment of the topic and we describe a practical implementation of a library search system, so that the present paper can be used as a reference both by people interested in the formal theory of ML style languages, and by people that is simply concerned with implementation issues. The new isomorphism can also be used to extend the usual ML type-inference algorithm, as suggested in (DC92). Building on that proposal, we introduce a better type-inference algorithm that behaves well in the presence of non-functional primitives like references and exceptions. The algorithm described here has been implemented easily as a variation to the Caml-Light 0.4 system.
KEYWORDS: library searches, types, isomorphisms, ML, type-assignment.

Capsule Review

This paper develops a complete proof system for type isomorphism in a type-assignment framework and gives a decision procedure. The main contribution of the paper is to set up a formalism for this proof system, to discover a new isomorphism (*split*), and to develop an efficient algorithm to check isomorphism, which can be used to improve library search systems, as investigated by Rittri and others. DiCosmo contributes to a better understanding of type isomorphism in ML-style type systems.

1 Foreword

This paper is devoted to the study of isomorphisms of types for type-assignment calculi based on Milner's ML (Mil78; Dam85; DM82; MT91; CH88; MTH90). It

§ Part of the material appearing in this paper has been published in the Proceedings of the 19th Symposium on Principles of Programming Languages, Albuquerque, New Mexico. The author would like to thank the Association for Computing Machinery for allowing him to use that material in this paper.

presents the motivations of such a study, a guide to previous works, a theoretical treatment of isomorphic types, a description of a library search system modulo equality up to isomorphic types, and a proposal for an extension of the ML type-inference mechanism, both based on the new results presented in this paper. The main focus is here on *formalization* of previous work in the area, and we try to show how such a formal treatment allows us to discover unexpected isomorphisms, and to derive naturally from the formal proofs a decision procedure more efficient than the ones that were previously available, in an effort of providing a systematic approach.
For this reason, this work may contain more than what you are looking for: it can be read not only from the start to the end, and not necessarily in sequential order, but in several different ways, depending on the interests of the reader. If you are just looking for an introduction to the topic, read Section 2 and 3. If you are already familiar with the topic, and your interest is for the new theoretical results, you can browse through Section 4 and then focus on Section 5, 6 and the Appendix. If you are just looking for a guide to the implementation of a library search system, then Sections 6 is all you need to see. If you are looking for the extended type-inference algorithm, then Section 7 will be your focus point.

2 Introduction

The interest in building models satisfying specific isomorphisms of types (or domain equations) is a long standing one, as it is a crucial problem in the denotational semantics of programming languages. In the 1980s, though, 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 we will call them in the sequel. The seminal paper by Bruce and Longo (BL85) addressed then the case of pure first and second order typed λ-calculus with essentially model-theoretic motivations, but due to the connections between typed λ-calculus, cartesian closed categories, proof theory and functional programming, the notion of *valid isomorphism of types* showed up as a central idea that translates easily in each of those different but related settings. In the framework of category theory, Soloviev already studied the problem of characterizing types (objects) that are isomorphic in every cartesian closed category, providing a model theoretic proof of completeness for the theory $Th^1_{\times\mathbf{T}}$ we will see later on (actually (Sol83) is based on techniques used originally in (Mar72), while another different proof can be found in (MS90)). A treatment of this same problem by means of purely syntactic methods for a λ-calculus extended with surjective pairing and unit type was developed in (BDCL92), where the relations between these settings, category theory and proof theory, originally suggested by Mints, have been studied, and pursued further on in (DCL89). Finally, (DC91) provides a complete characterization of valid isomorphisms of types for second order λ-calculus with surjective pairing and terminal object type, that includes all the previously studied systems.
Meanwhile, these results were starting to find their applications in the area of Functional Programming, where the problem of retrieving functions in a library was showing up as an increasingly relevant issue: while the size of the function libraries

Language	Name	Type
ML of Edinburgh LCF	itlist	$\forall X.\forall Y.(X \to Y \to Y) \to List(X) \to Y \to Y$
CAML	list_it	"
Haskell	foldr	$\forall X.\forall Y.(X \to Y \to X) \to X \to List(Y) \to X$
SML of New Jersey	fold	$\forall X.\forall Y.(X \times Y \to Y) \to List(X) \to Y \to Y$
The Edinburgh SML Library	fold_right	$\forall X.\forall Y.(X \times Y \to Y) \to Y \to List(X) \to Y$

Table 1. *an example*

grows steadily (the standard library of CAML v.2.6 contains already more than 1000 user-level identifiers, for example), the tools generally available today to retrieve functions stored in a library are still nothing more than a prehistorical alphabetical index of identifiers, maybe with some facility to enable regular-expression searches (like in the CAML interpreter, see (CH88)), or some kind of thesaurus, useful when you have to find your way in an UNIX manual (the well known *-k* option of the *man* command).

But the name given to a function is left to the more or less original imagination of the programmer, so if you change system, you change dialect also: borrowing an amusing example from (Rit90b), if we look for a function that distributes a binary operation on a list, we can easily collect a nice amount of names: *itlist, list_it, foldr, fold* and *fold_right*, so that the rudimentary tools available to search the libraries dont help at all. If we are using strongly typed functional languages, though, the Propositions as Types paradigm just tells us that a type can be considered as a (partial) logical specification of a program, suggesting to use *the type* of a function as a search key in order to provide the programmer with a uniform and sensible tool to retrieve data in libraries. The *types*, with their logical counterpart, would provide the necessary standard language.

This simple, but rather new idea is the starting point of work done by Mikael Rittri ((Rit91), (Rit90a)), Colin Runciman and Ian Toyn ((RT91)) in this direction. They immediately notice how functions that we want to consider essentially the same turn out to be assigned pretty different types. Borrowing from (Rit90b), we can provide an example of this unpleasant phaenomenon, just by looking at the type that the *itlist - list_it - foldr - fold - fold_right* function is assigned in five different widely used languages based on the same polymorphic type discipline originally presented in Milner's ML (Mil78) (see Table 1).

The syntactic equality of types is too much a fine relation on types to be used for our purposes: so what is the *right* way to compare types? We need a coarser relation on types that take into account, for example, currying-uncurrying and argument permutation. Moreover, this notion of equivalence ought not depend on the particular implementation of the language, and it needs to be decidable in order to be of any use.

We can clearly see the connection with the notion of type isomorphism described

above: for any typed functional language L, the equivalence relation on types will be exaclty the one given by the notion of *valid* isomorphism.

Definition 2.1 (Valid isomorphisms)

$A \cong B$ is a *valid isomorphism* $\Longleftrightarrow$ for **any** M model of L, $M \models A \simeq B$, i.e.

$$\exists f : A \to B,\ g : B \to A \text{ s.t. } g \circ f = id_A,\ f \circ g = id_B.$$

What is needed then is the ability to search types up to such isomorphisms, i.e. a *complete and decidable* characterization of the *valid* isomorphisms. The completeness of the theory is desirable, as a sound theory that is incomplete would miss part of the functions in the library.
In this paper, we survey the known results on valid isomorphisms of types (Section 3) and we point out why they are not adequate to handle languages where the *let* polymorphic construct is allowed. We study thereafter in Section 4 the problem of valid isomorphisms *in the presence* of such a polymorphic construct, and we uncover an isomorphism that does not hold for second order explicitly typed λ-calculus. This new isomorphism allows us to provide a complete and decidable characterization for ML like languages (Section 5). Furthermore, it makes it possible to design an efficient decision procedure for the complete theory of isomorphic types, that can be effectively used in an actual library search system. We provide a description of this procedure in Section 6.1, while the code for the CAML dialect of ML is publicly avilable by anonymous ftp (see Section 6.1 for details).
This new isomorphism can also be used to extend the usual ML type-inference algorithm, as proposed in (DC92). Building on that proposal, in Section 7 we introduce a more complete type-inference algorithm that is based on the notion of *inference-isomorphism* and behaves well in the presence of non-functional primitives like references and exceptions. The algorithm described here has been implemented easily as a variation to the Caml-Light 0.4 system.
Proofs of technical results are to be found in the Appendix.

3 Survey

In this section we survey the known results about the valid isomorphisms of types for first and second order λ-calculi, and we build up the necessary machinery to handle valid isomorphisms in type-assignment systems with the *let* construct. Since the focus of the paper is on the type-assignment systems, though, we do not give here the full syntax of the explicitly typed systems, but the interested reader can find a fully detailed presentation in (CDC91).

3.1 First order isomorphic types

In (BL85), Bruce and Longo showed that two types A and B are isomorphic in every model of the simply typed λ-calculus $\lambda^1\beta\eta$ if and only if they can be shown equal in the equational theory Th^1 that has only the following proper axiom

$$(\textbf{swap}) \quad A \to (B \to C) = B \to (A \to C)$$

$$\left.\begin{array}{l}(\textbf{swap}) \quad A \to (B \to C) = B \to (A \to C)\end{array}\right\} Th^1$$

$$\left.\begin{array}{ll}
1. & A \times B = B \times A \\
2. & A \times (B \times C) = (A \times B) \times C \\
3. & (A \times B) \to C = A \to (B \to C) \\
4. & A \to (B \times C) = (A \to B) \times (A \to C) \\
5. & A \times \mathbf{T} = A \\
6. & A \to \mathbf{T} = \mathbf{T} \\
7. & \mathbf{T} \to A = A
\end{array}\right\} Th^1_{\times \mathbf{T}}$$

$$\left.\begin{array}{ll}
8. & \forall X.\forall Y.A = \forall Y.\forall X.A \\
9. & \forall X.A = \forall Y.A[Y/X] \quad (X \text{ free for } Y \text{ in } A,\ Y \notin FTV(A)) \\
10. & \forall X.(A \to B) = A \to \forall X.B \quad (X \notin FTV(A))
\end{array}\right\} + \textbf{swap} = Th^2$$

$$\left.\begin{array}{ll}
11. & \forall X.A \times B = \forall X.A \times \forall X.B \\
12. & \forall X.\mathbf{T} = \mathbf{T}
\end{array}\right\}$$

$$\Big\} Th^2_{\times \mathbf{T}}$$

A, B, C can be arbitrary types and **T** is a constant for the unit type.
Notice that the axiom **swap** of Th^1 is provable in $Th^1_{\times \mathbf{T}}$ by axioms 1 and 3.

Table 2. *The theories of valid isomorphisms for explicitly typed languages*

where A, B, C can be arbitrary types.
A key point in the proof of completeness is the fact, very easy to show, that *valid* isomorphisms are always *definable* by programs in the language, i.e.

Proposition 3.1 (Definable isomorphisms)

A $\cong$ B $\Longleftrightarrow$ there exist λ-terms M : $A \to B$ and N :$B \to A$ such that $\lambda^1\beta\eta \vdash M \circ N = \mathrm{I_B}$ *and* $\lambda^1\beta\eta \vdash N \circ M = \mathrm{I_A}$, where $\mathrm{I_A}$ and $\mathrm{I_B}$ are the identities of type A and B, and M$\circ$N is the usual composition of terms λx.M(Nx). Such terms M and N are each other inverses w.r.t. composition, and are called "invertible".

This result holds for any of the languages we will survey in this section (see (DC91) for details), so we will talk indifferently about *valid* or *definable* isomorphisms, or just about isomorphisms.

Remark 3.2
Notice that we are in an explicitly typed framework, so the isomorphism between type A and B is given by explicitly typed terms M : $A \to B$ and N : $B \to A$.

Later on, this approach was extended to the lambda calculus with surjective pairing and terminal object ($\lambda^1\beta\eta\pi*$). Now, this calculus has as models exactly the Cartesian Closed Categories, that is a further reason for the relevance of the problem studied: in (Sol83) it is actually considered from the category theoretic point of view, and solved by model theoretic methods that can essentially be traced down to work done in number theory by Martin in (Mar72). A completely new argument based on proof theoretic methods was provided by Bruce, Longo and the author

(see (BDCL90)), where the connections with proof theory and functional programming are also outlined. The notion of isomorphism between types presented there is exactly the same adopted by Rittri in the case of ML-style languages, to the study of which he devotes several works ((Rit91), (Rit90a) and (Rit92)).
The resulting fundamental theorem in (Sol83) and (BDCL90) states that two types A and B are isomorphic in every model of the calculus $\lambda^1\beta\eta\pi*$ if and only if they can be shown equal in the equational theory $Th^1_{\times\mathbf{T}}$ of Table 2.

3.2 Second order isomorphic types

These results can be extended to second order typed λ-calculus, as in (BL85), where Bruce and Longo characterized the valid isomorphism for the pure second order λ-calculus $\lambda^2\beta\eta$ via the equational theory Th^2 of Table 2.
This result is not powerful enough, though, to treat ML-style systems, as we miss the product and the unit type constructors, so we need to look at (DC91), where a finite, decidable axiomatisation of the isomorphisms holding in the models of second order lambda calculus with surjective pairing and terminal object $\lambda^2\beta\eta\pi*$ is provided. The Main Theorem of that paper shows that two types A and B can be constructively proved to be isomorphic, by programs which act one as the inverse of the other, if and only if $Th^2_{\times\mathbf{T}} \vdash \mathrm{A} = \mathrm{B}$, where $Th^2_{\times\mathbf{T}}$ is the set of axioms in Table 2. This last theory of valid isomorphisms contains all the previous theories, and is the largest theory of isomorphic types in explicitly typed languages for which we have soundness and completeness results by now.

4 Isomorphisms of types in ML-style languages

In (Rit91) and (Rit90a), Rittri uses the theory $Th^1_{\times\mathbf{T}}$ to develop a library search system for strongly typed functional languages in the ML family. Languages of the ML family are equipped with the so-called implicit type polymorphism, a brand of type polymorphism that essentially allows to give the user the safety of a strongly typed world without the burden of mandatory type declarations: the user writes type-free programs and the compiler infers a type for it by filling in all the type information.
The inference problem is easily decidable in the case of monomorphic languages, like the simply typed λ-calculus, (see (Hin69), (Mil78)), while we do not know how to deal with it for calculi with the full power of second order quantification over types, like second order typed λ-calculus.
It is a common idea (but we will shortly see how it is not a very correct one) that ML-style languages lie somewhere in between these two extremes, as any user-defined function is given a type that can be more than monomorphic, but not fully second order polymorphic. These types are either monomorphic types (known as *monotypes* and denoted by τ below)† or the so-called type-schemes (denoted by σ below):

† The word *monotype* has been used in the literature with different meanings: it is to be noted that here a *monotype* is just a quantifier free type, that can contain type variables.

Definition 4.1
ML types are the *closed* types generated by the following grammar (At is a collection of atomic types)

$$\begin{array}{lll} \textit{type-schemes} & \sigma & ::= \tau \mid \forall X.\sigma \text{ (if X is free in } \sigma) \\ \textit{monotypes} & \tau & ::= At \mid X \mid \tau \to \tau \mid \tau \times \tau \end{array}$$

Type-schemes are essentially types where every type variable is bound by a quantifier that can appear only as an outermost constructor of the type (and not inside $\to$, $\times$ or other type contructors).
If we follow the common intuition that ML is somewhere in between simple typed λ-calculus and second order λ-calculus, it is easy to conjecture that the valid isomorphisms of type-schemes are axiomatized by a theory Th^{ML} that includes $Th^1_{\times\mathbf{T}}$ and is included in $Th^2_{\times\mathbf{T}}$.
Then, noticing that Axioms 10, 11 and 12 involve second order types that are not type-schemes, it seems reasonable that Th^{ML} be just $Th^2_{\times\mathbf{T}}$ less these three axioms. So a simple approach to deciding equality of type-schemes $\sigma_1 = \forall\overrightarrow{X}.\tau_1$ and $\sigma_2 = \forall\overrightarrow{Y}.\tau_2$, would be to check if there is a way of substituting in some order the variables $\overrightarrow{X}$ with $\overrightarrow{Y}$ in τ_1 such that for the resulting type τ_1' the theory $Th^1_{\times\mathbf{T}}$ proves $\tau_1' = \tau_2$. But in principle the restriction of $Th^2_{\times\mathbf{T}}$ to ML types is not necessarily axiomatised by the restriction to ML types of the axiomatic presentation $Th^2_{\times\mathbf{T}}$ we have chosen for this equality relation. Even worse, the techniques used to show completeness for $Th^2_{\times\mathbf{T}}$ on second order types rely in an essential way on the fact that the language considered there is explicitly typed, while ML-style languages are type assignment systems equipped with a *let* construct whose typing rules have no immediate counterpart in the explicitly typed calculi. So we could expect to find some isomorphism that is not axiomatised even in the full theory $Th^2_{\times\mathbf{T}}$.
Rittri's system (see (Rit91)), based on the well known soundness of $Th^1_{\times\mathbf{T}}$ for monomorphic languages, implements the procedure sketched above, and is *sound* for isomorphisms in ML, but to handle the *completeness* problem in ML we have to face the problem of valid type-schemes isomorphisms in its own right. It turns out that we are in for some surprises, here, but first of all, let's set up the right formalism for type-assignment systems.

4.1 A formal setting for valid isomorphisms in ML-like languages

Let's first briefly recall the basic typing rules for ML-like languages. We use pretty standard notation for λ- terms, but maybe for $FTV(A)$, that denotes the free type variables occurring in A, and the expressions p_1M and p_2M that stand for the first and second projection of a term M:

Definition 4.2 (Type assignment)
We write $\Gamma \vdash M : A$ if M *can* be assigned type A in the type assignment system given in Table 3.

Remark 4.3

$(UNIT)$ $\quad \Gamma \vdash () : \mathbf{T}$

(VAR) $\quad \Gamma \vdash x : \sigma[\tau_i/X_i]$ if $x : \sigma = \forall X_1 \ldots X_n.\tau$ is in Γ and the τ_i are monotypes

(ABS) $\quad \dfrac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x.M : \tau_1 \to \tau_2}$

(APP) $\quad \dfrac{\Gamma \vdash M : \tau_1 \to \tau_2 \quad \Gamma \vdash N : \tau_1}{\Gamma \vdash (MN) : \tau_2}$

$(PAIR)$ $\quad \dfrac{\Gamma \vdash M : \tau_1 \quad \Gamma \vdash N : \tau_2}{\Gamma \vdash < M, N > : \tau_1 \times \tau_2}$

$(PROJ)$ $\quad \dfrac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \mathrm{p}_i M : \tau_i} \qquad i = 1, 2$

(LET) $\quad \dfrac{\Gamma \vdash N : \tau_1 \quad \Gamma, x : \forall X_1 \ldots X_n.\tau_1 \vdash M : \tau_2}{\Gamma \vdash (\lambda x.M)N : \tau_2} \qquad (\{X_1 \ldots X_n\} \text{ is } FTV(\tau_1) - FTV(\Gamma))$

Table 3. *Type inference rules for an ML-like functional language.*

Notice that the (LET) rule gets priority on the ordinary (APP) rule, that is to say, if we have to type an application (MN) we use rule (LET) if M is a lambda abstraction, and rule (APP) otherwise.

In the traditional presentations, one avoids the overlapping of rules (LET) and (APP) by introducing the notation *let x = e' in e* for *(λx.e)e'*, but it is important to remark that this new notation is just syntactic sugar.
In what follows we will use indifferently *let x = e' in e* or *(λx.e)e'*, at our best convenience.
In the type-assignment framework, the Definition 2.1 used to introduce the notion of valid isomorphism is no longer appropriate: the programs we work with are assigned not only one, but several types, and we must take this fact into account. Indeed, the whole point of Definition 2.1 was to relate two types A and B when they admit bijective conversion functions; now, in explicitly typed systems, a function that can take an A into a B has exactly the type $A \to B$, but in a type assignment framework it is no longer so for two reasons:

- A or B can be now quantified types, and in our ML-like systems we do not have types like $A \to B$ in that case
- due to the let typing rule, given a function M with most general type $A \to B$, and an object O with most general type A, the application (MO) can have, in principle, most general type *strictly more general than* B.

We proceed then as follows.

Definition 4.4
We say that A and B are isomorphic w.r.t. the context Γ ($\Gamma \vdash A \cong B$) via M, M^{-1} iff using the typing rules given in Definition 4.2 the following holds

- $\forall P, \Gamma \vdash P : A \quad \Rightarrow \quad \Gamma \vdash (MP) : B \quad$ and $\quad \Gamma \vdash M^{-1}(MP) = P : A$

- $\forall Q, \Gamma \vdash Q : B \quad \Rightarrow \quad \Gamma \vdash (M^{-1}Q) : A \quad$ and $\quad \Gamma \vdash M(M^{-1}Q) = Q : B$

Notice that in the empty context all empty types‡ are vacuously isomorphic: for such types the premiss of the implication in the definition of isomorphism in the empty context cannot be satisfied, so the implication holds vacuously. This is one reason why $\emptyset \vdash A \cong B$ is *not* a good choice for the notion of isomorphism we need. Furthermore, we look for a notion of a *uniform* isomorphism, that does not depend on the context, in the sense that it works in *all* contexts, not just in the empty one. So we are led to the following

Definition 4.5 (ML isomorphism)
We say that A and B are isomorphic ($A \cong B$) via M, M^{-1} iff $\forall \Gamma, \Gamma \vdash A \cong B$ via M, M^{-1}.

It is an easy consequence of this definition the fact that M and M^{-1} are invertible, that is to say, $M \circ M^{-1} = \lambda$x.x and vice-versa, so it is not necessary to require this property explicitly.

Remark 4.6 (Closed terms)
With this definition, the only terms M, M^{-1} that can prove an isomorphism $A \cong B$ are closed: this comes from the fact that such terms must work in *any* context, so cannot have any free variables.

Now we can easily verify that Axiom 12 is in a sense still valid.

Remark 4.7
Let A be $\forall X.\sigma$, where σ is isomorphic to $\mathbf{T}$ via M, M^{-1}. Then it is easy to check that M, M^{-1} provide an ML-isomorphism between $\forall X.\sigma$ and $\mathbf{T}$ also.

So we must already add to our tentative definition of the Th^{ML} theory the following new Axiom **unit**, that is essentially Axiom 12 of $Th^2_{\times\mathbf{T}}$ restricted to ML types. This fact supports our original idea that Th^{ML} is more than just $Th^2_{\times\mathbf{T}}$ less Axioms 10, 11 and 12.

$$(\mathbf{unit}) \quad \forall X.A = \mathbf{T} \quad \text{if A is isomorphic to } \mathbf{T}$$

What comes in more as a surprise is that we also get a new isomorphism, not derivable in $Th^2_{\times\mathbf{T}}$, that originates from the peculiar typing rule used to obtain the traditional *let* polymorphism in ML-style languages.

Proposition 4.8
In ML-like languages, the following isomorphism hold

$$(\mathbf{split}) \quad \forall X.A \times B \cong \forall X.\forall Y.A \times (B[Y/X])$$

Proof
It suffices to provide M and M^{-1} s.t. $\forall \Gamma, \Gamma \vdash A \cong B$ via M, M^{-1}.
Let M be $\lambda x.\langle \mathrm{p}_1 x, \mathrm{p}_2 x\rangle$ and M^{-1} be λx.x, and let's check the conditions of Definition 4.4. Since these are closed terms, the context Γ poses no problem and it

‡ A type A is called *empty* if there is no closed term M of type A, i.e. no closed M s.t. $\emptyset \vdash M : A$.

is easy to check that, given N s.t. $\Gamma \vdash N : \forall X.A \times B$, we can derive, using the let polymorphic type inference rule, the following typing

$$\Gamma \vdash (\lambda x.\langle \mathrm{p}_1 x, \mathrm{p}_2 x\rangle)N : \forall X.\forall Y.A \times (B[Y/X])$$

Furthermore, $(\lambda x.x)((\lambda x.\langle \mathrm{p}_1 x, \mathrm{p}_2 x\rangle)N)$ can clearly be assigned type $\forall X.A \times B$.
For the other direction of the isomorphism, observe that, given N s.t. $\Gamma \vdash N : \forall X.\forall Y.A \times (B[Y/X])$, then, by instantiating both X and Y to X, we can derive

$$\Gamma \vdash (\lambda x.x)N : \forall X.A \times B$$

Now we can give back to the term $(\lambda / x.\langle \mathrm{p}_1 x, \mathrm{p}_2 x\rangle)((\lambda x.x)N)$ the original type $\forall X.\forall Y.A \times (B[Y/X])$ using again the let polymorphic type inference rule. □

Notice that there is an implicit side condition on the variable Y: it must not be free in B. Indeed, whenever applied to a type scheme, **split** can rename at will the bound type variables occurring in a product type, but it must not identify two different type variables in B.
Well, if you really don't believe it, just run your favorite typed functional language and try the following example (syntax of CAML):

Example 4.9

```
#let id x =x;;
Value id is <fun> : 'a -> 'a

#let double x = (x,x);;
Value double is <fun> : 'a -> 'a * 'a

#let join = double id;;
Value join is (<fun>,<fun>) : ('a -> 'a) * ('a -> 'a)

#(fun (f,g) -> (f,g)) join;;
(<fun>,<fun>) : ('a -> 'a) * ('b -> 'b)
```

□

Notice that the above example works in CAML, which gives the same type to an expression `(fun x -> e1) e2` as to the equivalent `let x = e2 in e1`, just like the inference rules of table 4.2, while in SML, for example, you have to explicitly tell the system about polymorphism by writing the last line as follows

Example 4.10

```
#let (f,g) = join in (f,g);;
(<fun>,<fun>) : ('a -> 'a) * ('b -> 'b)
```

□

Remark 4.11
The isomorphism **split** is not derivable in $Th^2_{\times \mathbf{T}}$.

Indeed, **split** allows to change the number of free type variables even in types that are not isomorphic to the unit type $\mathbf{T}$, while all the axioms in $Th^2_{\times \mathbf{T}}$ preserve that number for such types. This fact shows that type-assignment systems allow to prove *constructively* equivalent some proofs that are not so in the explicitly typed versions of the calculus, even at a higher order: actually, take the terms that are

(1)	$\lambda x.\langle \mathrm{p}_1 x, \mathrm{p}_2 x\rangle$	:	$A \times B$	$\cong$	$B \times A$	: $\lambda x.\langle \mathrm{p}_1 x, \mathrm{p}_2 x\rangle$
(2)	$\lambda p.\langle\langle \mathrm{p}_1 p, \mathrm{p}_1 \mathrm{p}_2 p\rangle, \mathrm{p}_2 \mathrm{p}_2 p\rangle$	:	$A \times (B \times C)$	$\cong$	$(A \times B) \times C$	: $\lambda p.\langle \mathrm{p}_1 \mathrm{p}_1 p, \langle \mathrm{p}_2 \mathrm{p}_1 p, \mathrm{p}_2 p\rangle\rangle$
(3)	$\lambda f.\lambda x.\lambda y.(f\langle x, y\rangle)$	:	$(A \times B) \to C$	$\cong$	$A \to (B \to C)$	: $\lambda f.\lambda p.(f \mathrm{p}_1 p)\mathrm{p}_2 p$
(4)	$\lambda f.\langle \lambda x.\mathrm{p}_1(fx), \lambda x.\mathrm{p}_2(fx)\rangle$	:	$A \to (B \times C)$	$\cong$	$(A \to B) \times (A \to C)$	: $\lambda p.\lambda x.\langle \mathrm{p}_1 px, \mathrm{p}_2 py\rangle$
(5)	$\lambda p.\mathrm{p}_1 p$	:	$A \times \mathbf{T}$	$\cong$	A	: $\lambda x.\langle x, ()\rangle$
(6)	$\lambda f.()$	:	$A \to \mathbf{T}$	$\cong$	$\mathbf{T}$	: $\lambda x.\lambda y.()$
(7)	$\lambda f.(f())$	:	$\mathbf{T} \to A$	$\cong$	A	: $\lambda x.\lambda y.x$
(8)	$\lambda x.x$	:	$\forall X.\forall Y.A$	$\cong$	$\forall Y.\forall X.A$	: $\lambda x.x$
(9)	$\lambda x.x$	:	$\forall X.A$	$\cong$	$\forall Y.A[Y/X]$	: $\lambda x.x$
(**split**)	$\lambda x.x$	:	$\forall X.A \times B$	$\cong$	$\forall X.\forall Y.A \times (B[Y/X])$	: $\lambda x.\langle \mathrm{p}_1 x, \mathrm{p}_2 x\rangle$
(**unit**)	f	:	$\forall X.A$	$\cong$	$\mathbf{T}$	: g if $f : A \cong \mathbf{T} : g$

A, B, C arbitrary types, **T** a constant for the **Unit** type
(in Axiom 9 X is free for Y in A, Y $\notin$ FTV(A)).

Table 4. *The isomorphisms of types for ML-like languages, and their realizers*§.

the natural candidates for proving (**split**) in $\lambda^2\beta\eta\pi*$ (they are *retyping functions*, in the terminology of (Mit88)):

$$M = \lambda p : (\forall X.A \times B).\lambda X.\lambda Y.\langle \mathrm{p}_1(p[X]), \mathrm{p}_2(p[Y])\rangle$$

$$M' = \lambda p : (\forall X.\forall Y.A \times B[Y/X]).\lambda X.\langle \mathrm{p}_1(p[X][X]), \mathrm{p}_2(p[X][X])\rangle$$

They provide just a retraction, and not an isomorphism: $M' \circ M$ is the identity on $\forall X.A \times B$, but $M \circ M'$ reduces to

$$\lambda z : (\forall X.\forall Y.A \times B[y/x]).\lambda X.\lambda Y.\langle \mathrm{p}_1(z[X][X]), \mathrm{p}_2(z[Y][Y])\rangle$$

which is in normal form, and not the identity.
So the original idea that ML is just a limited version of explicitly typed second order λ-calculus seems now to be a little less obvious: in (core) ML we cannot do everything we can do in explicitly polymorphic calculi, as it is well known, *but* it is also true that we can do in (core) ML something that cannot be done in the explicitly typed version of second order λ-calculus. Of course, it is to be noticed that if we take the type-assignment version of the second order λ-calculus, like the one used by (Kri90), then **split** becomes valid too: the erasure of the normal form of $M \circ M'$ above reduces to the identity with one step of Surjective Pairing. So it seems that the lesson to be learned here is that we need to be careful when using results from explicitly typed calculi in type-assignment frameworks and vice-versa.

5 Completeness and conservativity results

Are there any more unexpected isomorphisms coming out of the *let* construct? What about the Axioms 10 and 11 of $Th^2_{\times\mathbf{T}}$ we were forced to leave out? Do they induce some other derived isomorphisms on ML types?

§ Recall that we write $M : A \cong B : N$ when we want to make the realizers of an isomorphism explicit.

This is not the case, as we will see in a moment, and the Axioms we have found sound up to now are also complete, so we can finally give a name to our theory of ML isomorphisms.

Definition 5.1
Th^{ML} is the theory of equality defined by $Th^2_{\times\mathbf{T}}$ less Axioms 10, 11 and 12 plus **unit** and **split**.

We will present here two main results concerning Th^{ML}: one is completeness for ML isomorphisms, while the other shows how on ML types Th^{ML} is actually strictly more powerful than $Th^2_{\times\mathbf{T}}$.
Since the details of the proofs are rather technical, we postpone them to the Appendix, where the interested reader can find also the necessary technical definitions and references to previous related work. We provide here just the statement of the theorems, with a short sketch of the arguments of the proofs.

5.1 Completeness

The theory Th^{ML} can be shown complete by adapting to the type assignment framework the techniques introduced in (BDCL90). We first define a *split normal form* (see Definition 6.3) of types suggested by the Axioms of Th^{ML}, and we notice that completeness for ML isomorphisms reduces to completeness for isomorphisms of *split normal forms*. Then we provide a suitable notion of reduction on ML terms (Definition A.11) that is compatible with type assignment (Theorem A.12) and allows to study the invertible terms associated to these latter isomorphisms: we can find a syntactic characterization of such terms (Proposition A.21). This characterization is suitable to show completeness of Th^{ML} by induction on (roughly) the complexity of these invertible terms.

Theorem 5.2
The theory Th^{ML} is complete for ML isomorphisms.

Proof
See Theorem A.23 in Appendix. □

5.2 Relating $Th^2_{\times\mathbf{T}}$ and Th^{ML}

As for the relation between $Th^2_{\times\mathbf{T}}$ and Th^{ML}, a careful analysis of the invertible terms in $\lambda^2\beta\eta\pi*$ allows to show that **split** and **unit** give us back *the full power* of $Th^2_{\times\mathbf{T}}$ on ML types.

Proposition 5.3
Let A and B be ML types. If $Th^2_{\times\mathbf{T}}$ proves A = B, then Th^{ML} proves A = B too.

Proof
See Theorem A.27 in Appendix. □

To use standard terminology, one could say that on ML types the theory $Th^2_{\times\mathbf{T}}$ is *conservative* over Th^{ML}. Usually, though, when a conservativity result is stated,

it refers to some theory Th' that extends a theory Th but does not prove more equations on the language of Th. This is not the case here: since **split** is not derivable in $Th^2_{\times\mathbf{T}}$ (Remark 4.11), the theory Th^{ML} is strictly more powerful then $Th^2_{\times\mathbf{T}}$ on ML types. The previous proposition actually states that, on the class of ML types, Th^{ML} is an *extension* of $Th^2_{\times\mathbf{T}}$, and not the reverse, as one could have expected.

6 Deciding ML isomorphism

The proof of completeness allows to derive an easy decision algorithm for valid isomorphisms of ML types based on a variant of the *narrowing* technique. Every type A is first rewritten to a (unique) type normal form n.f.(A) via a strongly normalizing confluent type rewriting system derived from the axioms of Th^{ML} (it is a sub-system of the one used in (DC91), see Proposition 3.5 there).

Definition 6.1
(Type rewriting R) Let $\rightsquigarrow$ be the transitive and compatible type-reduction relation generated by:

$$\begin{array}{ll} A \times (B \times C) \rightsquigarrow (A \times B) \times C & \mathbf{T} \times A \rightsquigarrow \mathrm{A} \\ (A \times B) \to C \rightsquigarrow A \to (B \to C) & A \to \mathbf{T} \rightsquigarrow \mathbf{T} \\ A \to (B \times C) \rightsquigarrow (A \to B) \times (A \to C) & \mathbf{T} \to A \rightsquigarrow \mathrm{A} \\ A \times \mathbf{T} \rightsquigarrow \mathrm{A} & \forall X.\mathbf{T} \rightsquigarrow \mathbf{T}. \end{array}$$

Remark 6.2
A type normal form $n.f.(A)$ of a type A is just a type $\forall\overrightarrow{X}.(A_1 \times \ldots \times A_n)$, where no product or unit type appear in the A_i. We call the A_i the *coordinates* of A.

Then the presence of **split** suggests a further elaboration up to another normal form (no longer unique).

Definition 6.3
The *split-normal-form* s.n.f.(A) of a type A is obtained from n.f.(A) by applying as far as possible **split** to rename generic type variables.

Remark 6.4
As *split-normal-form* of a type A we can take a type of the form

$$\forall\overrightarrow{X_1}\ldots\overrightarrow{X_n}.(A_1 \times \ldots \times A_n),$$

where no product ot unit type appears in the A_i, no two A_i share generic type variables, and the $\overrightarrow{X_i}$ are *exactly* the free type variables of the A_i, noted $FTV(A_i)$.

In Appendix A, we will show that Th^{ML} proves A = B iff s.n.f.(A) is proven equal to s.n.f.(B) via associativity and commutativity of product, bound variable renaming (Axiom 9), quantifier swap (Axiom 8) and the derived Axiom **swap**.
This provides us with a very simple algorithm to decide equality in Th^{ML}.

Theorem 6.5 (Decidability of Th^{ML})
The theory of ML isomorphisms Th^{ML} is decidable.

Proof
Given types A and B, first reduce them to their split normal forms s.n.f.(A) and s.n.f.(B).
Now, associativity, commutativity of product, **swap** and quantifier swap do not change the length nor the alphabet of type expressions, so that equality up these Axioms is trivially decidable. Furthermore, associativity and commutativity of product and **swap** can be applied independently of variable renaming or quantifier swap, **swap** can be applied independently of associativity and commutativity of product and variable renaming and quantifier swap can be interchanged in a proof, so that any proof of equality

$$s.n.f.(A) = A_1 = \ldots = A_n = s.n.f.(B)$$

of the two split normal forms can be transformed, by reordering the axioms used, in a proof

$$s.n.f.(A) = A'_1 = \ldots = A'_n = s.n.f.(B)$$

that uses associativity and commutativity of product, **swap** and quantifier swap only after the other axioms, and we can restrict without loss of generality to proofs having this shape: a prefix where only variable renaming is used to prove s.n.f.(A) equal to some type expression A', and then a proof of A' = s.n.f.(B) where we have in this order quantifier swap, associativity and commutativity of product and **swap**.
Unfortunately, variable renaming can change the alphabet of a type expression, so it is potentially dangerous as it generates an infinite class of equal formulae, so the prefix of the equality proofs can be of arbitrary length endangering decidability, but it is actually harmless in this context: we are interested only in those proof prefixes that rename s.n.f.(A) to a A' that contain exactly the type variables of s.n.f.(B), as the rest of the proof cant change name of variables, and there is only a finite number of such renamings.
These observations give us the decision procedure: for every possible variable renaming σ (just $n!$ where n is the number of type variables in s.n.f.(A)), for every possible permutation of the coordinates (there are $n!$ if the coordinates are n) check for equality componentwise up to **swap** (this requires another factorial step). Succeed if you can find a variable renaming and a permutation of coordinates that provide componentwise equality. Fail otherwise. □

But what about *efficient* decidability? The simple algorithm used to prove decidability seems to imply a high complexity. We will describe here an improved decision procedure as it is implemented in the CAML system.

6.1 An Improved Decision Procedure

A careful analysis of the steps performed in the proof of decidability of Th^{ML} can help to develop a much more efficient algorithm, that can be used in a practical implementation.
Our algorithm, involving the simple steps described in the rest of this section, has been successfully implemented and is now part of the CAML system (the french

implementation of Milner's ML (CH88; WAL+90)). The interested reader will find a fully documented implementation, together with some historical remarks, on the anonymous ftp server `nuri.inria.fr`¶, but we could not resist presenting here the real code at least for the first step of the algorithm: indeed, in the CAML system the availability of a user level grammar to describe expressions and types of the language itself makes the implementation task very easy and the resulting code very elegant. A type can be described with the usual concrete syntax: it just suffices to declare it as a type expression (which is recognized by the grammar `gtype`) to the system by quoting it with `<:gtype< >>`.

Rewriting Types

The first step of our algorithm, following the line of our proof, will be to actually rewrite types to a *split-normal form*. In CAML, we can declare that the standard grammar will be `gtype` (the grammar for `type`s) so our CAML code for the type rewriting function can be defined by cases almost exactly as in Definition 6.1.

```
#set default grammar gtype:gtype;;

let rec rew_type =
  function
  | <<^x * ^y>>  -> rew_type_irr <<^(rew_type x) * ^(rew_type y)>>
  | <<^x -> ^y>> -> rew_type_irr <<^(rew_type x) -> ^(rew_type y)>>
  |          x   -> x
and rew_type_irr =
  function
    <<^x -> unit>>      -> <<unit>>
  | <<unit -> ^x>>      -> x
  | <<^x * unit>>       -> x
  | <<unit * ^x>>       -> x
  | <<(^x * ^y) -> ^z>> ->
      rew_type_irr <<^x -> ^(rew_type_irr <<^y -> ^z>>)>>
  | <<^x -> (^y * ^z)>> ->
      <<^(rew_type_irr <<^x -> ^y>>) * ^(rew_type_irr <<^x -> ^z>>)>>
  | x                   -> x ;;

type Type_Coords == int * gtype list;;

let TypeRewrite t = (flatten (rew_type t))
where rec flatten = function
    <<^x * ^y>> -> let (lgt1,l1) = flatten x and (lgt2,l2) = flatten y
                   in (lgt1+lgt2,append l1 l2)
  |       x     -> (1,[x]);;
```

Then we procede to rename the variables in the coordinates of a type: finally, rewriting a type to split normal form is accomplished by first rewriting it in the first order system, then renaming the variables as necessary.

```
let SplitTR typ = split_vars (TypeRewrite typ);;
```

¶ The search package has been added in directory `user_lib/FIND_IN_LIB` of the CAML distribution, that is in `/lang/caml/V3.1` on the server.

The function `split_vars` splits type variables in a type (represented as a coordinate list) by renaming them consecutively (further details are given and documented in the full implementation).
Next, we start by having another, different look at the problem of deciding equality in Th^{ML}, to discover that it can usefully be considered a special case of equational unification.

6.2 Equality as Unification with Variable Renamings

Since order and name of quantified type variables is irrelevant (Axioms 8 and 9), we can consider the problem of deciding

$$\forall \overrightarrow{X}.(A_1 \times \ldots \times A_n) \ = \forall \overrightarrow{Y}.(B_1 \times \ldots \times B_n)$$

in the last subtheory consisting of Axioms 1, 2, 8, 9 and **swap** as a special case of unification of

$$(A_1 \times \ldots \times A_n) = (B_1 \times \ldots \times B_n),$$

where we are not allowed to substitute arbitrary types for the type variables $\overrightarrow{X}$ and $\overrightarrow{Y}$, but just other type variables, with the constraint of not identifying variables that were originally different. Essentially, we restrict to unifiers that are just *bound variable renamings*. We will also call them in the following *consistent* variable renamings.
Again, two split normal forms are equal iff for some permutation $\sigma : n \to n$ their coordinates A_i and $B_{\sigma(i)}$ can be unified modulo **swap** with a variable renaming.
Unification up to **swap** (left-commutativity of $\to$) is decidable (see (Kir85)), so we can perform the necessary unification modulo **swap** for all permutations, and then check if there exists a permutation where unification succeeds with variable renamings.

6.2.1 Divide and Conquer

Actually, since all the variables are distinct in the different components, the result of unification on A_i and $B_{\sigma(i)}$ for a given permutation σ is completely independent of the outcome of unification on the other coordinates: the variable renaming we are looking for is actually made up of n independent variable renamings (one for each coordinate), so we can use a standard quadratic test to check only the $\frac{n(n+1)}{2}$ relevant permutations instead of trying all the $n!$ possible ones.
This is a significant cut-down on the number of coordinates checking: even without adopting dynamic programming techniques, we can see that the complexity goes steeply down from a monstruous $m!n!S$ that corresponds to trying equality modulo **swap** (of cost S) for all permutations of m variables and all permutation of n coordinates to an average (still fearful, but much lower) $n^2(\frac{m}{n})!S$ that corresponds to testing equality up to **swap** for each relevant permutation of coordinates and each permutation of the (average) $\frac{m}{n}$ type variables in a coordinate.
But there is still room for improvements.

6.3 Dynamic Programming

We can now try to attack also the complexity of checking variable-renamings. Instead of the naive approach consisting in, first, generation of all possible variable renaming, and then checking equality up to **swap**, we can use our knowledge that the needed variable renaming will have to satisfy equality up to **swap** to significantly cut down the number of renamings to generate and test.
Actually, any variable occurring rightmost in A cannot be moved by left commutativity, and must be renamed to a corresponding variable in B occurring in the same position. Any variable in rightmost position provides a part of the renaming that we look for, and rules out all the $(n-1)!$ renamings that do not agree. For example, when trying to show equal

$$A \to B \to X = A' \to B' \to Y,$$

we know that X must be associated with Y, so we need not try renamings that dont do this.
In unification up to left commutativity one takes this fact into account by using suitable flat normal forms (Kir85), where all permutable subformulas are flattened into a list, and the only rightmost non permutable subformula is singled out.
The unification procedure scans this data structure using all the variables in unmovable positions to build partial renamings, and stops as soon as an inconsistent variable renaming is reached (for example, as soon as the same variable is forced to be identified to more than one other variable). Anyway, when such inconsistencies are not encountered, and when we find variables whose binding is not determined, it is necessary to examine all possible permutations of the flat premisses list, and the associated renamings.
Our algorithm tries to adopt as much as possible dynamic programming techniques: we keep the current tentative variable renaming, and we fail as soon as it is made inconsistent by variable bindings imposed by the unification procedure. A renaming becomes inconsistent as soon as a same variable gets bound to more than one other variable, and one can check for this event while updating the variable renaming.
The basic steps of the unification procedure are as follows:

- **Check** if their flat lists have the same length and fail if it is not the case: **swap** does not change the length of flat lists.
- **Unify the heads**, that cannot be moved, and build a partial renaming.
- **Unify the flat lists** of premisses starting from the partial renaming built during the unification of the heads.

In case of failure, we restore the partial variable renaming to the state before the call to unification, in order to allow the backtracking that is necessary to perform unification of the premisses lists.
Once we are able to perform unification up to left commutativity, we are almost done: we just need a standard quadratic test to check equality of two lists of coordinates representing a type, and then we can put all together to get a function that tests for equality modulo ML isomorphisms.
Having this function, we can build a filter to be applied to the CAML system table in search for identifiers satisfying a type query.

As remarked in (Rit92), the rule $A \to \mathbf{T} = \mathbf{T}$, equates too much functions: in ML a function with only does side effects usually returns type $\mathbf{T}$, so that the premiss A is relevant to spot the behaviour of the function. For this reason, it is advisable to also provide a routine that does not implement the axiom $A \to \mathbf{T} = \mathbf{T}$ (this is done in our CAML implementation).

6.4 Experimental results

An experimental implementation of the improved algorithm has been performed in CAML (CH88) for the functional library of more than 1000 user identifiers that is available in the system. No preprocessing of the library has been performed, so that the reduction to normal form is repeated for every library identifier on every call to the search procedure, even if this work could be done once for all in a future stable version of the library search module.
Here are same examples of usage with their performance. The machine used to perform tests is a DECstation 5000 running CAML V3.1.

```
#timers true;;
() : unit

#search_iso "('a ->'b ->'b) * 'b * 'a list -> 'b";;
it_list : ('b -> 'a -> 'b) -> 'b -> 'a list -> 'b
list_it : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
() : unit
Evaluation has needed: Runtime: 3.14s GC: 0.96s
```

This first execution time includes the loading time for the search module.

```
#search_iso "('a -> 'b) -> 'a list -> 'b list";;
map : ('a -> 'b) -> 'a list -> 'b list
rev_map : ('a -> 'b) -> 'a list -> 'b list
map_succeed : ('a -> 'b) -> 'a list -> 'b list
() : unit
Evaluation has needed: Runtime: 1.06s GC: 0.91s

#search_iso "int * 'a list -> 'a";;
nth : 'a list -> int -> 'a
item : 'a list -> int -> 'a
() : unit
Evaluation has needed: Runtime: 1.05s GC: 0.88s

#search_iso "string*int -> string";;
first_n_string : int -> string -> string
last_n_string : int -> string -> string
following_word : string -> int -> string
() : unit
Evaluation has needed: Runtime: 1.05s GC: 0.88s

#search_iso "('a -> 'b * 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a";;
it_pair_list : ('a -> 'b * 'c -> 'a) -> 'a -> 'b list * 'c list -> 'a
list_it2 : ('b * 'c -> 'a -> 'a) -> 'b list -> 'c list -> 'a -> 'a
```

```
it_list2 : ('a -> 'b * 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a
() : unit
Evaluation has needed: Runtime: 1.01s GC: 0.88s

#search_iso "('a -> 'a) -> 'a list -> 'a list";;
map_share : ('a -> 'a) -> 'a list -> 'a list
share_map_share : ('a -> 'a) -> 'a list -> 'a list
() : unit
Evaluation has needed: Runtime: 1.01s GC: 0.88s

#search_iso "('a * 'b -> 'c) -> 'a -> 'b -> 'c";;
C : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c
uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
() : unit
Evaluation has needed: Runtime: 1.00s GC: 0.88s
```

7 Adding isomorphisms to the ML type checker

Up to now, we have been interested in isomorphisms of types just from the point of view of library searches using types as search keys, that is to say, we used such isomorphisms "after the fact": such a point of view is not the only interesting one, especially in the case of type-inference languages. One could be tempted to modify the very basic mechanism of the language, the type-inference algorithm, to incorporate these isomorphisms of types, in such a way that functions with isomorphic types can be simply interchanged, with the improved algorithm taking care of inserting the correct type conversion terms where and when necessary.
Now, it is doubtful if the isomorphisms in $Th^1_{\times\mathbf{T}}$ ought to be made part of the type-inference algorithm of an ML-style language essentially for two reasons:

- **Correctness:** the witnesses of the isomorphisms in $Th^1_{\times\mathbf{T}}$ do *change* the original program, so that the intended meaning of the program is not necessarily preserved when the program type-checks up to isomorphisms, but not in the original system. An easy example is the interaction of the commutativity of product on equal types with functions that are not commutative, like subtraction on numbers. There are ways to recover this particular case: one can either rule out commutativity completely, as done in (Nip90) (but then type errors deriving from erroneous argument order are not avoided), or one can try to control very careful the use of commutativity, forbidding it only on isomorphic types. Anyway, the whole matter is not clear enough to suggest such a modification right now.

- **Complexity:** adding all these isomorphisms at the type-inference level would probably require unification up to $Th^1_{\times\mathbf{T}}$, which is known to be undecidable (see (NPS92) for recent results), and even equality up to Th^{ML} is at least as hard as Graph Isomorphism (like in the case for $Th^2_{\times\mathbf{T}}$, see (Bas90) for details), so such a modification of the ML type-inference algorithm seems to be not feasible.

But if we look closer at **split**, we notice that there is something special in it w.r.t. the other isomorphisms: the terms that witness this isomorphism are essentially the identity. The invertible terms associated to *all* the other isomorphisms perform a coding that is simple, but does something to the term, while this is not so in the case of λx.x and $\lambda x. < \mathrm{p}_1 x, \mathrm{p}_2 x >$.
Indeed, the only interesting effect of the term $\lambda x. < \mathrm{p}_1 x, \mathrm{p}_2 x >$ is to allow the use of the *let* polymorphism necessary to change the type of the original term. This fact suggests that (**split**) has more to do with the type-inference algorithm than with the notion of *coding* we found at the basis of the equivalences needed in library searches performed on the basis of the type seen as a search key. So our concern about correctness of the transformation of program induced by the isomorphisms is no longer there if we consider (**split**) alone: there is *no* transformation of programs, so the intended meaning is surely preserved. We simply type check more programs, and we will see in a moment that the new program we allow to type-check *should already type-check*. As for complexity, we will propose below a straightforward modification of the type-inference rules that includes (**split**) at a very reasonable cost.
It is time for a working example: let's see *the same program* in ML that type checks only if written "the right way", while with **(split)** it would type-check in any case. Since it seemingly cracks the ML type checker, we will call the following program *crack*.

Example 7.1

```
   CAML (mips) (V 2-6.1) by INRIA Fri Nov 24 1989

#let join = let pair x = (x,x)
            in let id x = x
               in pair id;;
Value join = (<fun>,<fun>) : (('a->'a)*('a->'a))

#let split = let f x = x in (f,f);;
Value split = (<fun>,<fun>) : (('a->'a)*('b->'b))

#let crack f x y = ((fst f) x, (snd f) y);;
Value crack = <fun> : (('a->'b)*('c->'d)->'a->'c->'b*'d)

(* crack on split and different types *)

#crack split 3 true;;
(3,true) : (num * bool)

(* crack on join and different types *)

#crack join 3 true;;

line 1: ill-typed phrase, the constant true of type
bool cannot be used with type instance num in
crack join 3 true
1 error in typechecking

Typecheck Failed
```

□

Both functions, `join` and `split`, define a pair of identity functions, but only the split version survives the test of the context `crack _ 3 true`!
We can try to understand better what is going on by getting rid of the *let* construct via the usual translation `let x = e' in e` $\Rightarrow$ `(λx.e) e'`.

- `join` translates to
  $(\lambda pair.(\lambda f.pair\, f)(\lambda x.x))(\lambda x. < x, x >)$
- `split` translates to $(\lambda f. < f, f >)(\lambda x.x)$

Now it is easy to see what is going on: `join` and `split` translate to two terms that are not syntactically equal, but only up to the usual β conversion. Actually, `join` β-reduces to `split`.
Now, let's recall the key idea in *let* polymorphism: the polymorphic rule allows to give a type to an application if this application is typable in the monomorphic system *after one step of evaluation*. That is to say, to type (λx.M)N, we change the type-inference algorithm, that would try to give a type to (λx.M) and N separately, and only if it succeeds it tries to type their application. Instead, we look forward *just one* step of reduction, that is to say, we try to give a type to M[N/x]: if we succeed, that will be the type the original expression (λx.M)N will be given.
Well, `crack split 3 true` is *two* steps from `crack join 3 true`, so the original form of polymorphic type inference cannot get it! Adding **(split)** corresponds in a sense to moving forward more than one step in the type-inference process.

Remark 7.2
Of course there are lots of terms that are typable in the monomorphic discipline only after some steps of reductions, but the examples that are usually given typically involve a non typable subterm that is erased during these steps of reduction. For example, (λx.λy.y)Ω, where Ω is a diverging term, is of course not typable, while its reduct λy.y trivially has a type.
It is important to notice that this is *not* the case of `split` and `join`, as no interesting subterm is erased during the two steps of reductions that separate them.

So adding (**split**) to the type-checker is not just one of the various possible extensions of ML that can be suggested, but in a sense is a necessary completion of a language that allows, as it is now, one way of defining a pair of identity functions, while forbidding another that seems as perfectly correct.

*7.1 A modified type inference algorithm featuring just (*split*) polymorphism.*

We can easily modify the polymorphic type inference algorithm to accommodate (**split**) in the type-inference phase: it is just a matter of taking into account the renaming of type variables allowed by this axiom in the polymorphic type inference rule. So it is enough to add to the original ML type-inference algorithm the rule *split-let* of Table 5, with priority on the original *let* one.
This type checking algorithm assigns to `join` the same type as `split`, thus preventing the type error we saw in Example 7.1 above.

The original let inference rule

$$(\textbf{let})\quad \frac{\Gamma\vdash N : A \quad \Gamma, x : Gen(A)\vdash M : C}{\Gamma\vdash(\lambda x.M)N : C} \qquad \begin{array}{l}\text{Gen(A)} = \forall X_1 \dots X_n.A \text{ where} \\ \{X_1 \dots X_n\} \text{ is } FTV(A) - FTV(\Gamma)\end{array}$$

The let inference rule modified as in (DC92)

$$(\textbf{split} - \textbf{let})\quad \frac{\Gamma\vdash N : A \quad \Gamma, x : SplitGen(A)\vdash M : C}{\Gamma\vdash(\lambda x.M)N : C}$$

- $\text{SplitGen}(A \times B) = \forall X_1 \dots X_n Y_1 \dots Y_m.A \times (B[Y_1 \dots Y_m / X_{i_1} \dots X_{i_m}])$, where $X_{i_1} \dots X_{i_m}$ are the type variables shared by A and B, and $Y_1 \dots Y_m$ are fresh type variables.
- SplitGen(A) = Gen(A) if A is not a product type.

Table 5. *Modifying the let inference rule: a first attempt.*

Adapting an existing type-checker to accommodate this further rule is rather easy: the necessity of checking for shared type variables in product types requires some care in the actual implementation, but there is no need for new, complex unification procedures.

*7.2 What is special in (*split*)*

Why is it possible to add seamlessly (**split**) to the type checker, while other isomorphisms pose problems? Now, (**split**) essentially allows to rename the generic variables of any type schema that has a product as the outermost type constructor, in such a way that the two factors of the product do not share any generic type variable, but there are two very crucial properties enjoyed by (**split**) that make it suitable for use in type-inference:

- it is **Identity Based:** as pointed out before, its realizer is *the identity*. This means that to convert a given program P from one type to another in the equivalence class of types modulo **split** we need only apply to it a program equivalent to the identity, that does *not* alter P in any way‖.

- it is an **Instantiation Isomorphism:** the left hand side of **split** is a *generic instance*, in the usual sense, of its right hand side, but not viceversa. This do provides a best representative in the equivalence classes of the types isomorphic via **split**: the most generic type, that is to say the one obtained by applying **split** as far as possible from left to right.

These facts suggest to extend the original ML type-inference algorithm in such a way that the principal type schema inferred for a term is also the most generic one w.r.t. (**split**), as we have done in the previous section. Anyway, while this extension is obviously sound (it is easy to verify that we preserve all the good properties of the

‖ As is not the case of **curry** and **uncurry**, that do modify the functional behaviour of programs.

original inference algorithm in (Dam85; Mil78)), it is not so sure that it is the only possible one. Actually, the two properties of **split** that make it a good candidate for extension of the type-checker can be assumed as a criterion. We will use it to select, among the known isomorphisms, which ones are suitable for being incorporated in the type-inference mechanism and which ones are best left to be used in library searches only.

Criterion 7.3 (inference-isomorphisms)
Any isomorphism of types that is identity based (i.e. its realizers are the identity) and that is an instanciation isomorphism (i.e. one side of the isomorphism is a (generic) instance of the other side) can be incorporated in the ML type-inference algorithm.

For this reason, we will call such isomorphisms *inference-isomorphisms*.
We try now to support this "criterion" and apply it to the isomorphisms known to hold in the case of ML. This will lead us to the discovery of some more inference-isomorphisms than **split**, and we will describe in the last section how to modify the existing type-inference algorithm to accomodate the new isomorphisms, both in the pure calculus and in presence of imperative features like reference types and polymorphic exceptions.
We think that any inference-isomorphism of types $M : A \cong B : N$ should be incorporated into the type-inference algorithm for ML-like languages and, in our opinion, the following facts strongly support this view.

Correctness and Coherence. If $M : A \cong B : N$, then any program P of type A can be transformed in a program (MP) with type B. Since M is a program equivalent to the identity (as our isomorphism is identity-based), (MP) does exactly what P did, and the program transformation is trivially correct: *inference-isomorphism do not harm*. But B is a type more general than A (since our isomorphism is an instantiation): *inference-isomorphisms improve the system* . Here is an example of such phenomenon, using **split**.
Consider the following way of defining a pair of identity functions (syntax of CAML)**.

```
#let pair x = (x,x);;
Value pair is <fun> : 'a -> 'a * 'a

#let id x = x;;
Value id is <fun> : 'a -> 'a

#let idpair = pair id;;
Value idpair is (<fun>,<fun>) : ('a -> 'a) * ('a -> 'a)
```

This is not the best type one could expect: in fact, applying **split** we can get a better one:

** Again, remember that the last line in this example must be written in SML as follows to get the right type:

```
#let f = idpair in (fst f, snd f);;
(<fun>,<fun>) : ('a -> 'a) * ('b -> 'b)
```

```
#(fun f -> (fst f, snd f)) idpair;;
(<fun>,<fun>) : ('a -> 'a) * ('b -> 'b)
```

Principal Type Schema. If one side of the isomorphism is an instance of the other, then the principal type schema (pts) property of ML-style languages is preserved: we just get a "more general" pts (as in the previous example).

7.3 Choosing the Right Isomorphisms

In Table 4 we have an axiomatization of the theory of ML-style isomorphism, together with the realizers associated with each axiom. We can start looking for possible combinations of the axioms that give raise to isomorphisms complying with our criterion. We first notice that Axioms 8 and 9 are not interesting to us: even if they are realized by the identity, neither side is more general than the other. Actually they just tell us that the order and name of generic variables in type-schemas are inessential, fact that we already know. Then, it is easy to see that the following combinations work.

- Axiom 2 (associativity of $\times$), 8 and **split** allow to extend **split** from pairs to n-tuples:

  $\forall X.A_1 \times \ldots \times A_n = \forall X X_2 \ldots X_n.A_1 \times A'_2 \times \ldots \times A'_n \quad \text{where } A'_i = A_i[X_i/X]$

- Axiom 4 (distributivity of $\rightarrow$), 8 and **split** allow to extend **split** to higher order in a controlled way: if $X \notin FTV(A_i)$ we can state that

  $\forall X.A_1 \rightarrow \ldots \rightarrow A_n \rightarrow (B \times C) = \forall XY.A_1 \rightarrow \ldots \rightarrow A_n \rightarrow (B \times (C[Y/X]))$

Now these new inference-isomorphisms can be implemented in the ML type-inference algorithm by smarter and smarter generalizations procedures: the first ones require to split all the shared variables in all components of a tuple type, and not just the two components of a product; the second ones require to split the varibles of tuples also on the right of an arrow type constructor, and not only when the product is the toplevel type constructor (as in **split**). Furthermore, these new isomorphisms can be combined again with themselves allowing more and more splitting of shared generic type variables, and the final picture we get is the following:

Proposition 7.4
Given an ML-type A, the most general type B isomorphic to it via an inference-isomorphism derived by Axioms 2, 4 and **split** can be computed inductively as SplitGenIso(A, $\emptyset$), with SplitGenIso(A, V) defined as follows.

- $SplitGenIso(A \times B,\ V) = SplitGenIso(A, V) \times SplitGenIso(B', V)$
  where B' is B with FTV(B) - V replaced by fresh type variables
- $SplitGenIso(A \rightarrow B,\ V) = A \rightarrow SplitGenIso(B, V \cup FTV(A))$
- $SplitGenIso(C,\ V) = C$ otherwise

Hence, we propose to extend the ML type-inference mechanism by adding on top of the ordinary generalization mechanism the greater generality provided via inference-isomorphisms by SplitGenIso, as follows.

$$(\mathbf{split-gen-let}) \quad \frac{\Gamma \vdash N : A \quad \Gamma, x : SplitGenIso(Gen(A), \emptyset) \vdash M : C}{\Gamma \vdash (\lambda x.M)N : C}$$

This modified rule for typing the *let* construct is the one that has been added to Caml-Light (Ler90; Mau91). It clearly subsumes the original one, so that all programs accepted by the original ML algorithm can still be typed. But programs like `pair id` above are given more general types and it is very easy to find programs that type-check only in the new system.

7.4 Right Isomorphisms in Impure context

In the existing implementations of ML, it is necessary to take into account the impure features that are usually supported: polymorphic reference types and exceptions. The original ML type-inference algorithm is not sound in the presence of such constructs, as explained in (MT91) pagg. 41-46: it needs to be restricted. In SML (MTH90), generic variables are divided into imperative (noted `'_a`) and applicative ones (noted `'a`), and a simple restriction on the generalization of imperative variables guarantees soundness. For our isomorphism-based extension, we have a similar restriction: since an imperative generic variable represents a shared piece of memory, it is unsound to instantiate it to different types, so SplitGenIso(,) must not split imperative generic type variables. A careful analysis shows that this is already guaranteed by the restrictions imposed on imperative variables in SML, combined with the restriction on applicability of (**split**) on the right of the arrow.

8 Related Works

Isomorphisms of types are proving more and more powerful tools to increase the power and usability of typed functional programming languages. As pointed out in (DC92), where most of the results proved here were first presented without much proof, isomorphisms of types can be used either in collaboration with a human user to perform library searches based on types, or automatically to improve existing type-checkers.

While the only other work that tries to incorporate type isomorphisms at the type checker level is (Nip90), after the seminal paper (Rit91), several new works have appeared recently, all dealing with the use of types as search keys: Mikael Rittri has studied the possibility of performing searches by using pattern matching of types modulo $Th^1_{\times\mathbf{T}}$ (Rit90a; Rit90b), and has proposed also a search mechanism based on unification of types modulo linear isomorphisms (Rit92), which Brian Matthews (Mat) has extended to Haskell's system of type classes, in which type variables can be restricted to range only over certain types. There have also been two attempts to use formal specifications as search keys; both use a plain type as search key before they try to check whether any retrieved functions satisfy the given specification (Mor91; RW91). For the types, (Mor91) uses matching modulo isomorphism, while (RW91) uses incomplete unification.

These proposal turn library search via type-isomorphism into a very flexible tool, and show how the interest in this topic is rapidly growing, but none of them takes

into account (**split**): our work shows how this new isomorphism proves to be useful in designing an efficient search algorithm for the case of type equality. It is hopefully the case that it can help reduce the complexity of matching or unification too.

9 Conclusions

This paper has a twofold purpose: on one hand, it completes all the proofs sketched in (DC92), thus providing a firm basis to the theory of ML isomorphisms, on the other, it focuses on the practical issues of library searches, developing in details the search algorithm implemented in the CAML system. This algorithm is more efficient than the ones described in the references, and takes full advantage of the new **split** isomorphism to achieve its performance.
The very same new isomorphism (**split**) leads us to a more consistent version of the original ML type checker due to Milner (Dam85; Mil78). Even if the suggested modification is a minor one, it surely points out how misleading can be a certain use of terms like *completeness* theorems for type inference algorithms: our result does not contradict *completeness* theorems like the one in (Dam85), which states the *completeness* of the inference-algorithm w.r.t the ML type-assignment rules, and not the *completeness* of these typing rules w.r.t. to some class of models.
It deserves to be noticed here that recent works on *parametricity* (introduced in (Rey83), but see (ACC93) for very recent results and bibliography), seem to shed some light on the apparent anomalous (**split**) isomorphism. We have already hinted that the real issue is more an "explicitly vs. implicitly" typed than a "polymorphic λ- calculus vs. ML" one: if we take the somewhat less known implicitly typed version of the polymorphic λ- calculus, then (**split**) becomes provable. What one would really like to see is a result bridging this gap between the implicit and explicit presentations: the implicit calculi seem to be able to say "more" than the explicit ones (like what happens in the case of (**split**) here). Actually, in (LMS92) a simple extension of the explicit polymorphic λ- calculus is proposed, consisting of a single, very natural axiom saying that the result of a function $f : \forall X.A$ does not depend on the type argument if X does not occur free in A. Such slight addition is enough to derive the (**split**) isomorphism in the new calculus. We may hope that the results presented here can provide further motivations and applications to the theory of parametricity.

Acknowledgments

I'm greatly indebted to my advisor, Giuseppe Longo, for his continuous encouragement, discussions and insights: he strongly motivated me to carry on all the work that supports the results presented in this paper.
Mikael Rittri not only inspired this work on library searches, but has been a constant and copious source of information on the ongoing work in this area.
I wish to thank Hubert Comon and Jean-Pierre Jouannaud, as well as all the working group of the LRI at the University of Orsay, for several unvaluable discussions: they helped and pushed me in the essential phases of the development of this work.

I am grateful to Pierre-Louis Curien for the long and fruitful cooperation on the study of $\lambda^2\beta\eta\pi*$.
The whole group working at INRIA at the CAML system has provided unvaluable support in the development of the code that implements the library search system, and I'm particularly grateful to Pierre Weis whose help was crucial in the integration of this new tool in the CAML system, and to Xavier Leroy, whose impressive programming skill led to a working experimental implementation of the new type-inference mechanism for CamlLight 0.4 in less than an afternoon.
Thanks finally to Pierre Cregut and Delia Kesner for several afternoons spent discussing all these matters, and to the anonymous referees for their very stimulating reports.

References

Martín Abadi, Luca Cardelli, and Pierre-Louis Curien. Formal parametric polymorphism. In *Ann. ACM Symp. on Principles of Programming Languages (POPL)*. ACM, 1993. To appear.

Henk Barendregt. *The Lambda Calculus; Its syntax and Semantics (revised edition)*. North Holland, 1984.

David Basin. Equality of Terms Containing Associative-Commutative Functions and Commutative Binding Operators is Isomorphism Complete in 10th Int. Conf. on Automated Deduction. *Lecture Notes in Computer Science*, 449, July 1990.

Kim Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. Technical Report 90-14, LIENS - Ecole Normale Supérieure, 1990. To appear in Proc. of Symposium on Symbolic Computation, ETH, Zurich, March 1990, *Mathematical Structures in Computer Science*, 2(2).

Kim Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2(2), 1992. Proc. of Symposium on Symbolic Computation, ETH, Zurich, March 1990.

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

Pierre-Louis Curien and Roberto Di Cosmo. A confluent reduction system for the λ-calculus with surjective pairing and terminal object. In Leach, Monien, and Artalejo, editors, *Intern. Conf. on Automata, Languages and Programming (ICALP)*, pages 291–302. Springer-Verlag, 1991.

Guy Cousineau and Gerard Huet. The caml primer. Technical report, LIENS - Ecole Normale Supérieure, 1988.

Luis Damas. *Types Disciplines in Programming Languages*. PhD thesis, Computer Science Dept., University of Edimburgh, April 1985.

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. Submitted to *Information and Computation*.

Roberto Di Cosmo. Type isomorphisms in a type assignment framework. In *Ann. ACM Symp. on Principles of Programming Languages (POPL)*, pages 200–210. ACM, 1992.

Roberto Di Cosmo and Giuseppe Longo. Constuctively equivalent propositions and isomorphisms of objects (or terms as natural transformations). *Workshop on Logic for Computer Science - MSRI, Berkeley*, November 1989.

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

Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Ann. ACM Symp. on Principles of Programming Languages (POPL)*, pages 207–212. ACM, 1982.

R. Hindley. The principal type-scheme of a an object in combinatory logic. *Transactions of the American Mathematical Society*, 146, 1969.

Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-calculus*. London Mathematical Society, 1980.

C. Barry Jay. Strong normalisation for simply-typed lambda-calculus as in lambek-scott. LFCS, University of Edimburgh, February 1991.

Claude Kirchner. *Methodes et utiles de conception systematique d'algoritmes d'unification dans les theories equationnelles*. PhD thesis, Université de Nancy, 1985.

Jean-Louis Krivine. *Lambda calculus. Types et Modéles*. Masson, 1990.

Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.

Giuseppe Longo, Kathleen Milsted, and Sejey V. Soloviev. The genericity theorem and the notion of parametericity in the polimorphic λ-calculus. E-mail: longo@di.ens.fr and milsted@prl.dec.com., August 1992.

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

Brian Matthews. Reusing functional code using type classes for library search. Dept. Comput. Sci, University of Glasgow, Scotland. E-mail: brian@dcs.glasgow.ac.uk.

Michel Mauny. *Functional programming using Caml Light*. INRIA, 1991. Included in the Caml Light distribution.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17(3):348–375, 1978.

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

R. Morgan. *Component Library Retrieval using property models*. PhD thesis, University of Durham - England, rick@easby.dur.ac.uk, 1991.

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

Robin Milner and Mads Tofte. *Commentary on Standard ML*. The MIT Press, 1991.

Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

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. E-mail: dran@cs.albany.edu, 1992.

J.C. Reynolds. Types, abstraction and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing '83*. North Holland, 1983.

Mikael Rittri. Retrieving library identifiers by equational matching of types in 10th Int. Conf. on Automated Deduction. *Lecture Notes in Computer Science*, 449, July 1990.

Mikael Rittri. *Searching program libraries by type and proving compiler correctness by bisimulation*. PhD thesis, University of Göteborg, Göteborg, Sweden, 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.

Colin Runciman and Ian Toyn. Retrieving re-usable software components by polymorphic type. *Journal of Functional Programming*, 1(2):191–211, 1991.

E. J. Rollins and J. M. Wing. Specifications as search keys for software libraries. In K. Furukawa, editor, *Eighth International Conference on Logic Programming*, pages 173–187. MIT Press, 91.

Serjey V. Soloviev. The category of finite sets and cartesian closed categories. *Journal of Soviet Mathematics*, 22(3):1387–1400, 1983.

Pierre Weis, María Virginia Aponte, Alain Laville, Michel Mauny, and Ascánder Suárez. The CAML reference manual. Technical Report 121, INRIA, Roquencourt B.P.105 - 78153 Le Chesnay Cedex - France, September 1990.

A Technical proofs.

This Appendix contains the proofs of Theorems 5.2 and Proposition 5.3, and the definitions of the technical notions needed for them. The interested reader should refer to (BDCL90; DC91) too.

A.1 Technical definitions and properties

We will use in what follows many notions for which we provide here all the relevant definitions. Anyway, the interested reader can find in (Bar84) a detailed account of the properties of *finite-hereditary-permutations* (f.h.p.'s) and *Böhm tree* (BT(M)) of a term M and in (BL85) and especially (DC91) the relevant facts about *second order finite-hereditary-permutations* (2-f.h.p.'s).

Definition A.1
The **Böhm-tree** BT(M) of a term M is (informally) given by:

$$\begin{array}{ll} \mathrm{BT(M)} = \Omega \text{ if M has no head normal form} & \\ \mathrm{BT(M)} = \quad \lambda \mathrm{x}_1 \ldots \mathrm{x}_n.y \quad & \text{if } \mathrm{M} =_\beta \lambda \mathrm{x}_1 \ldots \mathrm{x}_n.y\mathrm{M}_1 \ldots \mathrm{M}_p \\ \qquad\qquad /...\backslash & \\ \qquad BT(M_1) \ldots BT(M_p) & \end{array}$$

It is easy to observe that BT(M) is finite and Ω-free iff M has a normal form.

Definition A.2
[Finite Hereditary Permutations (f.h.p.)] Let M be an untyped term. Then M is a **finite hereditary permutation** (f.h.p.) iff either

- $\lambda^1\beta\eta \vdash \mathrm{M} = \lambda x.x$, or
- $\lambda^1\beta\eta \vdash \mathrm{M} = \lambda z.\lambda \overrightarrow{x}.z\overrightarrow{N}_\sigma$,
  where if $|\overrightarrow{x}| = n$ then σ is a permutation over n and $\mathrm{z}\overrightarrow{N}_\sigma = (\ldots(\mathrm{zN}_{\sigma(1)}) \ldots \mathrm{N}_{\sigma(n)})$, such that, for $1 \leq i \leq n$, $\lambda x_i.N_i$ is a finite hereditary permutation.

Thus $\lambda z.\lambda x_1.\lambda x_2.zx_2x_1$ and $\lambda z.\lambda x_1.\lambda x_2.zx_2\lambda x_3.\lambda x_4.x_1x_4x_3$ are f.h.p.'s.
F.h.p.'s possess normal forms that are closed terms. In particular, exactly the abstracted variables at level $n+1$ appear at level $n+2$, modulo some permutation of the order (note the special case of z at level 0). The importance of f.h.p.'s arises from the following theorem, where the notion of invertible term given in 3.1 easily translates to the untyped λ-calculus.

Theorem A.3 (Dezani (Dez76))
Let M be an untyped term possessing normal form. Then M is $\lambda_{\beta\eta}$-invertible iff M is a f.h.p.

Remark A.4

One may easily show that the f.h.p.'s are typable terms (Hint: Just follow the inductive definition and give z, for instance, type $A_1 \to (A_2 \ldots \to B)$, where the A_i's are the types of the $N_{\sigma(i)}$.) By the usual abuse of language we may then speak of typed f.h.p.'s.

When dealing with second order lambda calculus, we can get a similar characterization of invertible terms, where also types are allowed in the f.h.p.

Definition A.5
A second order term M of $\lambda^2\beta\eta$ is a second order finite hereditary permutation (2-f.h.p.) iff

- M = λx.x, or
- M = $\lambda z.\lambda\overrightarrow{v_i}.z\overrightarrow{P_i}$ where $z\overrightarrow{P_i}$ is $zP_1 \ldots P_n$ and, if $\mid \overrightarrow{v_i} \mid$ = n, there exists a permutation $\sigma{:}n \to n$, such that
  if $\lambda v_i = \lambda x_i$:C then λx_i:C.$P_{\sigma(i)}$ is a 2-f.h.p.
  if $\lambda v_i = \lambda X_i$ then $P_{\sigma(i)}$ is X_i.

Theorem A.6
2-f.h.p.'s are the invertible terms of $\lambda^2\beta\eta$.

Proof
See (BL85), Lemma 2.4 and Theorem 2.5. □

Finally, let's make it precise what is the formal system we consider here as (core) ML:

Definition A.7
[The ML formal system] Consider the untyped lambda terms generated by the following grammar:

$$t := ()\mid x\mid \lambda x.t\mid (tt)\mid \langle t,t\rangle\mid \mathrm{p}_1 t\mid \mathrm{p}_2 t$$

The formal system for (core) ML is made of the untyped lambda terms t that can be assigned a type in the type-assignment system given in Table 3.

A.2 Completeness

To show completeness of Th^{ML}, we first notice that each type reduction rule in $\rightsquigarrow$ (see Definition 6.1) derives from a valid isomorphism. So to each such type reduction is associated an isomorphism, and then, since isomorphisms compose, *any* isomorphism M can be decomposed as in Figure 1, where F and G, with their inverses F^{-1} and G^{-1}, are the isomorphisms associated to the rules used to rewrite the types A and B to their *split-normal-form*.
It is evident from the diagram that two types A and B are isomorphic iff their split normal forms are. Now, reduction to split normal form is done accordingly to some axioms of Th^{ML}, so that to prove completeness of this theory it suffices to prove completeness for isomorphisms between types in *split-normal-form*. In order to do this, we study the structure of a generic invertible term providing an isomorphisms between such types. We follow the techniques introduced in (BDCL90) and (DC91)

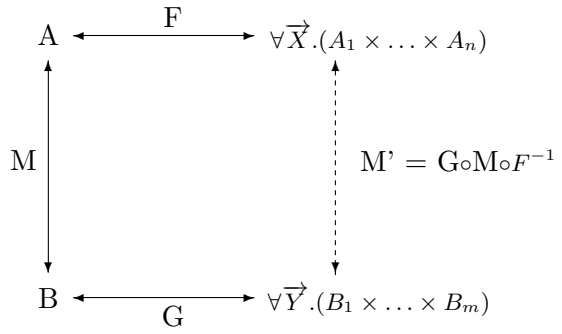

Fig. 1. Decomposition of an ML isomorphism.

for the case of explicitly typed languages, that we adapt here to the type assignment framework.
To deal with the strucure of terms we need to work on representatives of the equivalence classes of terms w.r.t. the equality theory of the calculus, such as a normal form. So we first need to provide a suitable notion of reduction that preserves (or at least does not decrease) the set of types that can be assigned to a term. This is not a concern in the case of explicitly typed languages, but in this type assignment framework it requires some care, as the following remark shows.

Remark A.8
The reduction rule for Surjective Pairing

$$(SP) \qquad \langle \mathrm{p}_1 M, \mathrm{p}_2 M\rangle \text{ reduces to } M$$

strictly decreases the set of types that can be assigned to a term by the type-inferernce algorithm of Definition 4.1.

Indeed, take this simple program that builds a pair of identity functions, then decomposes and builds it up again via projection and pairing.

Example A.9

```
#let splitpair =
     let  join = let pair x = (x,x) in let id x = x in pair id
#in (fst join, snd join);;
Value splitpair is (<fun>,<fun>) : ('a -> 'a) * ('b -> 'b)
```

□

Its most general type is `('a -> 'a) * ('b -> 'b)` and it would reduce, if we allow SP contraction, to the following

Example A.10

```
#let splitpair =
     let join  = let pair x = (x,x) in let id x = x in pair id
#in join;;
Value splitpair is (<fun>,<fun>) : ('a -> 'a) * ('a -> 'a)
```

□

Anyway, `('a -> 'a) * ('a -> 'a)` is not an instance of `('a -> 'a) * ('b -> 'b)`,

which is more general: we lost in the reduction the possibility to instantiate the two components of the product type to different types.
It is necessary to devise a notion of reduction that is compatible with the type assignment, i.e. that allows to prove a Subject Reduction theorem. Actually, if we orient (SP) the other way round, to get a Surjective Pairing Expansion as suggested for example in (Jay91), we get a normalizing calculus for which the reductum of a term M can be given at least all the types that are legal for M. Notice that, since we work in a type assignment framework, reductions are relativized by a basis Γ where the types of the free term variables are declared.

Definition A.11
(Notion of reduction for ML)

alpha-beta-eta-csi:

$(\rightarrow \beta)$ $\Gamma\vdash(\lambda$x.M)N $\stackrel{ML}{\longrightarrow}$ M[x:=N] : A, if N is free for x in M

$(\rightarrow \eta)$ $\Gamma\vdash\lambda$x.(Mx) $\stackrel{ML}{\longrightarrow}$ M : $A \rightarrow B$, if $x \notin FV(M)$

surjective pairing:

$(\times\beta)$ if $\Gamma\vdash\langle M_1, M_2\rangle : A_1 \times A_2$, $\Gamma\vdash \mathrm{p}_i(\langle M_1, M_2\rangle) \stackrel{ML}{\longrightarrow} M_i : A_i$

$(\times\eta)$ if $\Gamma\vdash$M:$A \times B$, $\Gamma\vdash$M $\stackrel{ML}{\longrightarrow}$ $\langle \mathrm{p}_1(M), \mathrm{p}_2(M)\rangle$:$A \times B$††

terminal object:

$(*)$ if $\Gamma\vdash$M : $\mathbf{T}$ then $\Gamma\vdash$M $\stackrel{ML}{\longrightarrow}$().

When talking about reduction, normal form and similar notion in what follows, we will refer to this notion $\stackrel{ML}{\longrightarrow}$ of reduction on ML terms.

Theorem A.12 (Subject reduction)
Let $\Gamma\vdash$M $\stackrel{ML}{\longrightarrow}$ M'. If $\Gamma\vdash$M:A, then $\Gamma\vdash$M':A.

Proof
Essentially the same as in (HS80), Theorem 15.17. □

The Subject Reduction theorem provides us immediately with a nice relation between typings derivable with or without the let rule.

Proposition A.13
Any *closed* term M that is in normal form can be turned into an explicitly typed term of $\lambda^1\beta\eta\pi*$.

Proof
Typing with the let rule corresponds to typing without the let rule after one step of parallel reduction, so, unless there are free variables with polymorphic types, on normal forms the two typing mechanism coincide. Since we do not need polymorphism to type M, and M has no free variables, we can decorate every lambda abstraction with the type in the derivation of $\emptyset\vdash$M : A and get a term of $\lambda^1\beta\eta\pi*$ with the same type. □

Remark A.14

Due to the Subject Reduction theorem, we can always consider that in A ≅ B via M, M^{-1}, the terms M and M^{-1} are in normal form.

Remark A.15
Isomorphisms do compose.

Now we can carry on our analysis of invertible terms. The key results in Section 3 of (BDCL92) essentially allow us to characterize the invertible terms of $\lambda^1\beta\eta\pi*$, and can be used indirectly to chracterize invertible terms for the type assignment case. We provide here the statements of these results, while the proofs are detailed in (BDCL92):they tell us that isomorphic types of $\lambda^1\beta\eta\pi*$ in *split-normal-forms* have the same number of coordinates, so that, in Figure 1, $n = m$. Furthermore, for any given isomorphism M between *split-normal-forms* there exist a permutation $\sigma : n \to n$ such that M can be split into componentwise isomorphisms M_i between A_i and $B_{\sigma(i)}$.

Proposition A.16 (Isolate the relevant $\langle M_1, \ldots, M_n\rangle$ in a $\lambda^1\beta\eta\pi*$ isomorphism)

Let $S \equiv \forall\overrightarrow{X}.\mathrm{S}_1 \times \ \ldots \ \times \mathrm{S}_m$ and $R \equiv \forall\overrightarrow{Y}.\mathrm{R}_1 \times \ \ldots \ \times \mathrm{R}_n$ be type-n.f.'s where neither the S_i's nor the R_j's contain any occurrences of $\mathbf{T}$ or $\times$. Then $S \cong R$ iff there exist $\mathrm{M}_1, \ldots, \mathrm{M}_n$ and $\mathrm{N}_1, \ldots, \mathrm{N}_m$ such that

$\mathrm{x}_1 : \forall\overrightarrow{X}.S_1, \ldots, \mathrm{x}_m : \forall\overrightarrow{X}.S_m \vdash \langle M_1 : R_1, \ldots, M_n : R_n\rangle : \mathrm{R}_1 \times \ \ldots \ \times \mathrm{R}_n$

$\mathrm{y}_1 : \forall\overrightarrow{Y}.R_1, \ldots, \mathrm{y}_n : \forall\overrightarrow{Y}.R_n \vdash \langle N_1 : S_1, \ldots, N_m : S_m\rangle : \mathrm{S}_1 \times \ \ldots \ \times \mathrm{S}_m$

$M_i[\overrightarrow{N}/\overrightarrow{x}] =_{\beta\eta\pi SP} y_i$, for $1 \le i \le n$ and $N_j[\overrightarrow{M}/\overrightarrow{y}] =_{\beta\eta\pi SP} x_j$, for $1 \le i \le m$

(where substitution of vectors of equal length is meant componentwise).

Proof
One uses the fact that

$$M = \lambda z.(\lambda x_1 \ldots x_n.M\langle x_1, \ldots x_n\rangle)(p_1 z)\ldots(p_n z)$$

Now, the type of $M\langle x_1, \ldots x_n\rangle$ is R, an n-tuple, so the right hand side term is also equal to

$$\lambda z.(\lambda x_1 \ldots x_n.\langle M_1, \ldots M_n\rangle)(p_1 z)\ldots(p_n z)$$

where the M_i (which are the normal forms of $\mathrm{p}_i M\langle x_1, \ldots x_n\rangle$), are the sought terms, as shown in Proposition 3.7 of (BDCL92) using the fact that M and N are each other's inverses. □

Lemma A.17 (Isomorphic type-n.f.'s have equal length)

Assume that $S \equiv \forall\overrightarrow{X}.\mathrm{S}_1 \times \ \ldots \ \times \mathrm{S}_m$ and $R \equiv \forall\overrightarrow{Y}.\mathrm{R}_1 \times \ \ldots \ \times \mathrm{R}_n$ are type-n.f.'s and $M \equiv \langle M_1, \ldots, M_n\rangle$, $N \equiv \langle N_1, \ldots, N_m\rangle$ are terms in $\lambda^1\beta\eta\pi$ such that

$\mathrm{x}_1 : \forall\overrightarrow{X}.S_1, \ldots, \mathrm{x}_m : \forall\overrightarrow{X}.S_m \vdash M_i : R_i \quad M_i[\overrightarrow{N}/\overrightarrow{x}] =_{\beta\eta\pi SP_{exp}} y_i$, for $1 \le i \le n$

$\mathrm{y}_1 : \forall\overrightarrow{Y}.R_1, \ldots, \mathrm{y}_n : \forall\overrightarrow{Y}.R_n \vdash N_j : S_j \quad N_j[\overrightarrow{M}/\overrightarrow{y}] =_{\beta\eta\pi SP_{exp}} x_j$, for $1 \le i \le m$

then $n = m$ and there exist permutations σ, π over n (and terms P_i, Q_j) such that

$$M_i = \lambda \overrightarrow{u_i}.x_{\sigma_i}\,\overrightarrow{P}_i \qquad \textit{and} \qquad N_j = \lambda \overrightarrow{v_j}.x_{\pi_i}\,\overrightarrow{Q}_j$$

Proof
This result is obtained by a short computation of $\lambda x.M(Nx)$ and $\lambda y.N(My)$, in Lemma 3.8 of (BDCL92). □

Furthermore, these terms M_i can be used to build componentwise isomorphisms between A_i and $B_{\sigma(i)}$.

Proposition A.18
Let $\mathrm{M}_1, \ldots, \mathrm{M}_n$ and $\mathrm{N}_1, \ldots, \mathrm{N}_n$ and permutation σ be as above. Then $\lambda x_{\sigma(i)}.M_i$ and $\lambda y_i.N_{\sigma(i)}$ are invertible terms.

Proof
Using Dezani's theorem (Dez76), one shows that each of the M_i, N_j contain exactly one free variable, and then it is easy to show that $\lambda x_{\sigma(i)}.M_i$ and $\lambda y_i.N_{\sigma(i)}$ are invertible terms (BDCL92). □

Now, to go back to the type assignment framework, we already know that, if $\mathrm{A} \cong \mathrm{B}$ via M, M^{-1}, and M is in normal form, then

- M is typable in $\lambda^1\beta\eta\pi*$, with the same type $A' \to B'$ it gets in ML
- M is invertible, and provides an isomorphism between A' and B'.

Using these facts, we obtain the following characterization:

Proposition A.19
Let $\mathrm{A} = \forall \overrightarrow{X}.(A_1 \times \ldots \times A_n)$ and $\mathrm{B} = \forall \overrightarrow{Y}.(B_1 \times \ldots \times B_m)$ be isomorphic *split-normal-forms* in ML. Then $n = m$ and there exist an invertible term M in normal form proving $\mathrm{A} \cong \mathrm{B}$ and a permutation $\pi : n \to n$ s.t.

$$M = \lambda z.\langle M_1[\mathrm{p}_{\pi(1)}z/x_{\pi(1)}], \ldots, M_n[\mathrm{p}_{\pi(n)}z/x_{\pi(n)}]\rangle$$

where $\lambda x_{\pi(i)}.M_i$ are f.h.p.'s.

Proof
The term M is typable in ML with some type $C \to D$, with D more general than $B_1 \times \ldots \times B_m$. Without loss of generality, we can assume that the invertible term M in ML providing the isomorphism is of type $C \to B_1 \times \ldots \times B_m$. Then it provides in $\lambda^1\beta\eta\pi*$ an isomorphism between C and $B_1 \times \ldots \times B_m$, so that by Proposition A.16 and A.17,

$$M = \lambda z.\langle M_1[\mathrm{p}_{\pi(1)}z/x_{\pi(1)}], \ldots, M_n[\mathrm{p}_{\pi(n)}z/x_{\pi(n)}]\rangle$$

where $\lambda x_{\pi(i)}.M_i$ are f.h.p.'s. In particular, this allows to conclude that $n = m$. □

Proposition A.13 allows us to prove immediately a nice result on arrow only types.

Corollary A.20
Let A, B be ML types without occurrence of products or unit type. If $\Gamma \vdash \mathrm{A} \cong \mathrm{B}$ via invertible terms M, M^{-1} in normal form, then it is possible to decorate M, M^{-1} with types and get terms $M'{:}A' \to B'$, $M'^{-1}{:}B' \to A'$ of $\lambda^1\beta\eta\pi*$, where A and B are just Gen(A') and Gen(B') up to generic type variable renaming.

Proof
By Proposition A.13, any type derivable for M, M^{-1}, which are *closed* (see Remark 4.6) is derivable without use of `let`, and M', M'^{-1} are then just the $\lambda^1\beta\eta\pi*$ terms of Proposition A.13, with types $A' \to B'$ and $B' \to A'$. Furthermore, since these terms are f.h.p.'s, any term variable occurring in them occurs exactly once and bound.
Now recall what $\Gamma\vdash$A $\cong$ B via M, M^{-1} means: for any term P s.t. $\Gamma\vdash$P:A, $\Gamma\vdash$(MP):B and $\Gamma\vdash M^{-1}(MP) = P : A$ (and viceversa).
Even if the `let` rule is not needed in typing M, it can be used in typing the application MP, as M (a f.h.p.) is actually a lambda abstraction. But the bound variable occurs only once in M, so the result of applying the let rule is nothing more than an instantiation of A to A' and a renaming and a generalization of B' to B. □

This result can actually be extended to all *split-normal-forms* by examining the structure of invertible terms that transform *split-normal-forms* to *split-normal-forms*.

Proposition A.21
Let A $= \forall\overrightarrow{X}.(A_1 \times \ldots \times A_n)$ and B $= \forall\overrightarrow{Y}.(B_1 \times \ldots \times B_n)$ be isomorphic *split-normal-forms*, of length n. Then there exist an invertible term M in normal form proving A $\cong$ B and a permutation $\pi : n \to n$ s.t.

$$M = \lambda z.\langle M_1[\mathrm{p}_{\pi(1)}z/x_{\pi(1)}], \ldots, M_n[\mathrm{p}_{\pi(n)}z/x_{\pi(n)}]\rangle$$

where $\lambda x_{\pi(i)}.M_i$ are f.h.p.'s that can prove $Gen(A_{\pi(i)})$ isomorphic to a variable renaming of $Gen(B_i)$.

Proof
We first obtain from M, using Proposition A.19 above the M_i such that $\lambda x_{\pi(i)}.M_i : A' \to B'$, then use Corollary A.20 to show the relation between A', B' and A, B. □

Remark A.22
The invertible term M in the previous Proposition A.21 is already in normal form (the only potential redexes are the occurrences of z that could be SP expanded, but the expansion is not allowed as z is under the action of a projection).

The following Completeness Theorem can now be shown as Proposition 4.8 of (BDCL90) by induction on the structure of the invertible terms.

Theorem A.23 (Completeness)
The theory Th^{ML} is complete for ML isomorphisms.

Proof
If two types A and B are isomorphic, then there is an invertible term M proving it of the form shown in Proposition A.21. Then one can show by induction on the Böhm tree of M that $Th^{ML} \vdash$A = B, as in Proposition 4.8 of (BDCL92).

□

A.3 Conservativity

Lemma A.24
Let M:$A \to B$ be a 2-f.h.p. (in normal form). If A and B are types not containing quantifiers, them M is a term of $\lambda^1\beta\eta$ (the simple typed λ-calculus) and Axiom **(swap)** suffices to prove A = B.

Proof
By induction on the depth n of the Böhm-tree BT(M) of M.

- $n = 1$.
  Then M = λx : A. x : $A \to A$ and A = B, so the thesis holds.
- $n = \mathrm{k} + 1$.
  Then M = λz : A.$\lambda\overrightarrow{v_i}$.z$\overrightarrow{P_i}$ where the v_i are all term variables whose type C_i does not contain $\forall$., since B does not contain $\forall$.. Now, we know from the definition of second order f.h.p. that the $\lambda v_i : C_i.P_{\sigma(i)}$ are second order f.h.p.'s for some permutation $\sigma : n \to n$. Furthermore, we know that the type D_i of each of the P_i has no occurrence of $\forall$. in it as otherwise A would have occurrences of $\forall$., in order for z$\overrightarrow{P_i}$ to type-check.
  Summing up, we know that the second order f.h.p. $\lambda v_i : C_i.P_{\sigma(i)}$ has type $C_i \to D_{\sigma(i)}$ with no occurrence of $\forall$., and its Böhm-tree has a strictly lower depth than BT(M). So we can apply the induction hypothesis and we get that each of these terms is just a simply typed term, hence M, which is built up out of them, is a simply typed term.
  Secondly, this proves that $Th^1 \vdash C_i = D_{\sigma(i)}$. But

$$A \equiv D_1 \to \ldots D_n \to E \quad \textit{and} \quad B \equiv C_1 \to \ldots C_n \to E$$

  for some base type E, in order for M to typecheck. So we get the second part of the thesis by:

$$\begin{aligned} B &\equiv C_1 \to \ldots C_n \to E \\ &= D_{\sigma(1)} \to \ldots D_{\sigma(n)} \to E(\text{by the induction hypothesis}) \\ &= D_1 \to \ldots D_n \to E(\text{by swapping the premisses}) \\ &\equiv A \end{aligned}$$

  where the last equality step uses several times the proper axiom of Th^1 in order to re-arrange the premisses of the $\to$ in the right order.

□

Theorem A.25
Let $\forall\overrightarrow{X}.A$ and $\forall\overrightarrow{Y}.B$ be second order types such that A and B do not contain quantifiers, products and the unit type. If $Th^2_{\times\mathbf{T}} \vdash \forall\overrightarrow{X}.A = \forall\overrightarrow{Y}.B$, then $Th^{ML} \vdash \forall\overrightarrow{X}.A = \forall\overrightarrow{Y}.B$.

Proof

Suppose that the given types are equal in $Th^2_{\times\mathbf{T}}$. They are already in normal form w.r.t. the rewriting system **R** of (DC91), Definition 3.4, so by Theorem 3.32 of (DC91) their isomorphism is witnessed by an invertible term M that is actually a 2-f.h.p. (a term of $\lambda^2\beta\eta$).
Now, $Th^2_{\times\mathbf{T}}$ does not allow to change the number of quantifiers in a type unless there is at least an occurrence of the unit type in their scope, and this is forbidden by our hypotheses, so we know that the length n of $\overrightarrow{X}$ is equal to that of $\overrightarrow{Y}$.
Knowing all this, let's study the term M. It is a 2-f.h.p., so (see (DC91), Definition 3.29)

$$M = \lambda z : (\forall\overrightarrow{X}.A).\lambda Y_1 \dots Y_n.\lambda x_{n+1} \dots x_{n+k}.zP_1 \dots P_{n+k}$$

In a 2-f.h.p., all the abstracted type variables must appear once and only once at the level immediately below that where they are abstracted, so, due to the type of z and the fact that A does not contain quantifiers, the first n P_i's must be exactly the type variables $\overrightarrow{Y}$ in some order. This means that, for the permutation $\sigma : n+k \to n+k$ associated to the 2-f.h.p. M, we have that $\lambda x_i.P_{\sigma(i)}$ are 2-f.h.p.'s whose types do not contain quantifiers (or otherwise, due to the fact that A does not contain quantifiers, M would not type-check). Hence the real structure of M is

$$\lambda Y_1 \dots Y_n \lambda x_{n+1} \dots x_{n+k}.z[Y_{\sigma(1)} \dots Y_{\sigma(n)}]P_{n+1} \dots P_{n+k},$$

where we know by Lemma A.24, that the 2-f.h.p.'s $\lambda x_i.P_{\sigma(i)}$ (and hence the P_{n+i}'s), are simple typed terms of $\lambda^1\beta\eta$.
Now, by a simple induction on the depth of the Böhm tree of M it is easy to show that $\forall\overrightarrow{X}.A = \forall\overrightarrow{Y}.B$ can be proved using only **(swap)** and Axioms 8 and 9, that are all derivable in Th^{ML}. □

Corollary A.26
Let $\forall\overrightarrow{X}.A$ and $\forall\overrightarrow{Y}.B$ be second order types as above in Theorem A.25. Let $\forall\overrightarrow{X'}.A$ and $\forall\overrightarrow{Y'}.B$ be the ML types obtained from them by erasing all quantifications on type variables not occurring in A and B respectively. Then $Th^2_{\times\mathbf{T}} \vdash \forall\overrightarrow{X}.A = \forall\overrightarrow{Y}.B$ $\Rightarrow Th^{ML} \vdash \forall\overrightarrow{X'}.A = \forall\overrightarrow{Y'}.B$

Proof
Suppose $Th^2_{\times\mathbf{T}} \vdash \forall\overrightarrow{X}.A = \forall\overrightarrow{Y}.B$.
The terms P_{n+i}'s and the variables x_i's in Theorem A.25 contain as free type variables only the $\overrightarrow{Y'}_i$'s, as only these variables occur in the type B, so we can build the term $M' = \lambda w : (\forall\overrightarrow{X'}.A).\lambda\overrightarrow{Y'}.\lambda x_{n+1} \dots x_{n+k}.w[Y'_\sigma]P_{n+1} \dots P_{n+k}$ where Y'_σ is what is left of $Y_{\sigma(1)} \dots Y_{\sigma(n)}$ after erasing the type variables not occurring in B.
The term M' type checks‡‡, and proves (in $Th^2_{\times\mathbf{T}}$) $\forall\overrightarrow{X'}.A = \forall\overrightarrow{Y'}.B$, so we can apply once more Theorem A.25 and finally get $Th^{ML} \vdash \forall\overrightarrow{X'}.A = \forall\overrightarrow{Y'}.B$, as required. □

‡‡ Notice that $\overrightarrow{Y'}$ and $\overrightarrow{X'}$ have the same length, since the rules in $Th^2_{\times\mathbf{T}}$ do not change the number of bound variables to prove $\forall\overrightarrow{X}.A = \forall\overrightarrow{Y}.B$

Theorem A.27
(Th^{ML} subsumes $Th^2_{\times\mathbf{T}}$ on ML types)
Let C and D be any ML types. If $Th^2_{\times\mathbf{T}} \vdash$ C = D, then $Th^{ML} \vdash$ C = D.

Proof
Let C = $\forall\overrightarrow{X}.A$, and C = $\forall\overrightarrow{Y}.B$ be ML types equated in $Th^2_{\times\mathbf{T}}$. Take their normal forms n.f.(C) and n.f.(D) w.r.t. the type rewriting system **R** of (DC91). We know that, since they are equal in $Th^2_{\times\mathbf{T}}$, there is an n s.t. n.f.(C) = $(C_1 \times \ldots \times C_n)$ and n.f.(D) = $(D_1 \times \ldots \times D_n)$, where no product or unit type appears in the C_i's and the D_i's. Moreover, the rewriting rules in R do not push any $\forall$ inside $\rightarrow$ or $\times$, and we start with ML-style types (that have $\forall$ only as the outermost type constructors), so we know that the C_i and the D_i are still ML-style types. More than that, we know that for some types A_i and B_i not containing quantifiers $C_i \equiv\forall\overrightarrow{X}.A_i$ and $D_i \equiv\forall\overrightarrow{Y}.B_i$. Now, Theorem 3.32 in (DC91) says that there exist a permutation $\sigma : n \rightarrow n$ s.t. for all i $Th^2_{\times\mathbf{T}}\vdash \forall\overrightarrow{X}.A_i = \forall\overrightarrow{Y}.B_{\sigma(i)}$. Let's call $\overrightarrow{X_i}$ and $\overrightarrow{Y_i}$ the type variables free in the A_i's and the B_i's respectively. Now Corollary A.26 states that $Th^{ML}\vdash\forall\overrightarrow{X_i}.A_i = \forall\overrightarrow{Y_{\sigma(i)}}.B_{\sigma(i)}$ Since we can rename bound type variables in Th^{ML}, these equalities can be turned into $Th^{ML}\vdash\forall\overrightarrow{X'_i}.A'_i = \forall\overrightarrow{Y'_{\sigma(i)}}.B'_{\sigma(i)}$ where all the type variables have been renamed in such a way that no two A'_i's or $B'_{\sigma(i)}$'s share any type variable. If M'_i's are the ML terms associated to these equalities in Th^{ML}, then we can build the ML term $\lambda w.\langle M'_1(\mathrm{p}_{\sigma(1)}w), \langle \ldots, M'_n(\mathrm{p}_{\sigma(n)}w)\rangle \ldots\rangle$ that proves

$$Th^{ML} \vdash \forall\overrightarrow{X'_1}\ldots\overrightarrow{X'_n}.(A'_1 \times \ldots \times A'_n) = \forall\overrightarrow{Y'_1}\ldots\overrightarrow{Y'_n}.(B'_1 \times \ldots \times B'_n)$$

These two last types are in normal form w.r.t the type rewriting system $\rightsquigarrow$, that is a subsystem of **R** in (DC91), and moreover all the coordinates have disjoint type variables: they are actually *split-normal-forms* of C and D.
Now, Th^{ML} proves that any ML type is equal to any of its *split-normal-forms* (see again Figure 1), so, by transitivity, $Th^{ML}\vdash$ C = D, as required. □

Remark A.28
Notice that the proof relies in an essential way on the equivalence between an ML type and its *split-normal-form*, that is due to Axiom **split**. Actually, without it, the previous theorem is false, as the following example shows.

Example A.29

Let A and B be type expressions that cannot be equated in $Th^2_{\times\mathbf{T}}$ nor in Th^{ML}. Then it is easily seen that

$$\begin{aligned} Th^2_{\times\mathbf{T}} \vdash\ & \forall XY.(X \rightarrow (X \rightarrow Y) \rightarrow A) \times (Y \rightarrow (Y \rightarrow X) \rightarrow B) \\ =\ & \forall ZW.(Z \rightarrow (Z \rightarrow W) \rightarrow B) \times (Z \rightarrow (Z \rightarrow W) \rightarrow A). \end{aligned}$$

But Th^{ML} without Axiom **split** *cannot prove it: these types are already in normal form w.r.t. $\rightsquigarrow$, and there is no way to equate them with only variable renaming, permutation or swapping of premisses.* □