Abstract

This paper contains a full treatment of isomorphic
types for languages equipped with an ML style poly-
morphic type inference mechanism. Surprisingly
enough, the results obtained contradict the common-
place feeling that (the core of) ML is a subset of sec-
ond 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 not
only allows to provide a complete (and decidable) ax-
iomatisation of all the types isomorphic in ML style
languages, a relevant issue for the type as specifica-
tions paradigm in library searches, but also suggest a
natural extension that in a sense completes the type-
 inference mechanism in ML. This extension is easy to
implement and allows to get a further insight in the
nature of the let polymorphic construct.

1 Introduction

The interest in building models satisfying specific iso-
morphisms of types (or domain equations) is a very
long standing one, as it is a crucial problem in the
denotational semantics of programming languages. Since
1984, 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 Pro-
gramming, 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 stud-
ied the problem of characterizing types (objects) that
are isomorphic in every cartesian closed category, pro-
viding a model theoretic proof of completeness for the
theory $Th^{h}_{T}$ we will see later on [Sol83]. A treat-
ment of this same problem by means of purely syn-
tactic methods for a λ-calculus extended with Surjec-
tive Pairing and unit type was developed in [BDCL90],
where the relations between these settings and Proof
Theory, originally suggested by Mints, have been stud-
ied, and pursued further on in [DCL89], where a new
model of typed λ-calculus is also proposed. Finally,
[DC91] provides a complete characterization of valid
isomorphisms of types for second order λ-calculus with
Surjective Pairing and unit 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 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 fa-
cility to enable regular-expression searches (like in the
CAML interpreter, see [CH88]), or some kind of thes-
saurus, 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 immediately notice how functions that are not essentially the same turn out to be assigned pretty different types. Borrowing from [Rit90a], we can provide an example of this unpleasant phaenomenon, just by looking at the type that the function is assigned

\[ \text{itlist}, \text{list_it}, \text{foldl}, \text{fold} \text{ and } \text{fold_left}, \]

so that the rudimentary tools available to search the libraries dont help at all. If we are using strongly typed functional languages, though, the Proofs 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 ([Rit89], [Rit90a]), Colin Runciman and Ian Toyn ([RT89]) in this direction. They immediately notice how functions that are not 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 - foldl - fold - fold_left` 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).

<table>
<thead>
<tr>
<th>Language</th>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML of Edinburgh LCF</td>
<td>itlist</td>
<td>( \forall X, Y. (X \to Y \to Y) \to \text{List}(X) \to Y )</td>
</tr>
<tr>
<td>CAML</td>
<td>list_it</td>
<td>&quot;</td>
</tr>
<tr>
<td>Haskell</td>
<td>foldl</td>
<td>( \forall X, Y. (X \to Y \to X) \to X \to \text{List}(Y) \to X )</td>
</tr>
<tr>
<td>SML of New Jersey</td>
<td>fold</td>
<td>( \forall X, Y. (X \times Y \to Y) \to \text{List}(X) \to Y \to Y )</td>
</tr>
<tr>
<td>The Edinburgh SML Library</td>
<td>fold_left</td>
<td>( \forall X, Y. (X \times Y \to Y) \to Y \to \text{List}(X) \to Y )</td>
</tr>
</tbody>
</table>

Table 1: an example on types will be exactly the one given by the notion of *valid* isomorphism.

**Definition 1.1** (*Valid isomorphisms*)

\( A \cong B \) is a *valid* isomorphism \( \iff \) for any \( M \) model of \( L \), \( M \models A \cong B \), i.e.

\( \exists f : A \to B, \ g : B \to A \text{ s.t. } g \circ f = \text{id}_A, \ f \circ g = \text{id}_B. \)

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 obviously essential, 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 2) and we point out why they are not adequate to handle languages where the *let* polymorphic construct is allowed. We study thereafter in Section 3 the problem of valid isomorphisms in the presence of such a polymorphic construct, and we provide a complete and decidable characterization for it in Section 4. This characterization uncovers a new and much unexpected isomorphism that does not hold for second order typed \( \lambda \)-calculus. It can be used to extend the usual ML type-inference algorithm, as proposed in Section 5. Finally, in Section 6 we summarize the key contributions of this paper and some open issues arising from this work.

**2 Survey**

In this section we survey the known results about the valid isomorphisms of types for first and second order \( \lambda \)-calculi, and we build up the necessary machinery to handle valid isomorphisms in type-assignment systems with the *let* constructor. For the full syntax of the typed calculi referred below, see [CDC91].

**2.1 First order isomorphic types**

In [BL85], Bruce and Longo showed that two types \( A \) and \( B \) are isomorphic in every model of the simple
Proposition 2.1 (Definable isomorphisms)

definable by programs in the language, i.e.
equal in the equational theory $T_{\lambda}$

following proper axiom

Explicitly typed terms $M : A \rightarrow B$ such that $\lambda^1 \beta\eta \vdash M \circ N = I_B$ and $\lambda^1 \beta\eta \vdash N \circ M = I_A$, where $I_A$ and $I_B$ are the identities of type $A$ and $B$, and $M \circ N$ is the usual composition of terms $\lambda x. M(Nx)$.

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 2.2**

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

Later on, this approach was extended to the lambda calculus with surjective pairing and terminal object $\lambda^1 \beta\eta\pi*$, i.e. the internal language of Cartesian Closed Categories. In [Sol83] this problem is solved by model theoretic methods that can essentially be traced down to work done in number theory by Martin ([Mar72]), while a completely new proof based on proof theoretic methods was provided by Bruce, Longo and the author (see [BDCL90]). 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 the two papers [Rit89] and [Rit90a].

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 $T_{\lambda T}$ of Table 2.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$A \times B = B \times A$</td>
<td>$\vdash_{T\lambda T}$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$A \times (B \times C) = (A \times B) \times C$</td>
<td>$Th^1_{\lambda T}$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$(A \times B) \rightarrow C = A \rightarrow (B \rightarrow C)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$A \rightarrow (B \times C) = (A \rightarrow B) \times (A \rightarrow C)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$A \times T = A$</td>
<td>$Th^2_{\lambda T}$</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$A \rightarrow T = T$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>$T \rightarrow A = A$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>$\forall X. \forall Y. A = \forall Y. \forall X. A$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>$\forall X. A = \forall Y. A[Y/X]$ $(X$ free for $Y$ in $A$, $Y$ not free in $A)$</td>
<td>$Th^2$</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>$\forall X. (A \rightarrow B) = A \rightarrow \forall X. B$ $(X$ not free in $A)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>$\forall X. A \times B = \forall X. A \times \forall X. B$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>$\forall X. T = T$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 $Th^2_{\lambda T}$ by axioms 1 and 3.

2.2 Second order isomorphic types

These results can be extended to second order typed $\lambda$-calculus, as in [BL85], where Bruce and Longo characterized the valid isomorphism for the pure second order $\lambda$-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 $\lambda^2\beta\eta\pi\ast$ 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 $Th_{xT}^2 \vdash A = B$, where $Th_{xT}^2$ is the set of axioms in Table 2. This last theory of valid isomorphisms contains all the previous theories and is as far as we can go by now.

3 Isomorphisms of types in ML-style languages

In [Rit89] and [Rit90a], Rittri uses the theory $Th_{xT}^1$ 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 $\lambda$-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 $\lambda$-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) (denoted by $\tau$ below) or the so-called type-schemas (denoted by $\sigma$ below):

**Definition 3.1** ML types are the closed types generated by the following grammar ($At$ is a collection of atomic types)

- type-schemas $\sigma ::= \tau \mid \forall X.\sigma$ (if $X$ is free in $\sigma$)
- monotypes $\tau ::= At \mid X \mid \tau \rightarrow \tau \mid \tau \times \tau$

Type schemas 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 $\rightarrow$, $\times$ or other type constructors).

If we follow the common intuition that ML is somewhere in between simple typed $\lambda$-calculus and second order $\lambda$-calculus, it is easy to conjecture that the valid isomorphisms of type-schemas are axiomatized by a theory $Th^{ML}$ that includes $Th_{xT}^1$ and is included in $Th_{xT}^2$.

Then, noticing that Axioms 10, 11 and 12 involve second order types that are not type-schemas, it seems reasonable that $Th_{xT}^{ML}$ be just $Th_{xT}^2$, less these three axioms. So the naive approach to deciding equality of type-schemas $\sigma_1 = \forall X.\tau_1$ and $\sigma_2 = \forall Y.\tau_2$, would be to check if there is a way of substituting in some order the variables $X$ with $Y$ in $\tau_1$ such that for the resulting type $\tau'_1$ the theory $Th_{xT}^1$ proves $\tau'_1 = \tau_2$. We say naive, because in principle the restriction of $Th_{xT}^2$ to ML types is not necessarily axiomatised by the restriction to ML types of the axiomatic presentation $Th_{xT}^2$ we have chosen for this equality relation. Even worse, the techniques used to show completeness for $Th_{xT}^2$ 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 $Th_{xT}^2$.

Rittri’s system (see [Rit89]), based on the well known soundness of $Th_{xT}^1$ 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-schema 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.

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

Let’s first briefly recall the basic typing rules for ML-like languages:

**Definition 3.2 (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 3.3** Notice that the *(LET)* rule gets priority on the ordinary *(APP)* rule: we do not introduce here the usual syntactic sugar let $x = e'$ in $e$ for $(\lambda x.e)e'$.

In this type-assignment framework, the Definition 1.1 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. We proceed as follows.

**Definition 3.4** We say that $A$ and $B$ are isomorphic w.r.t. the context $\Gamma$ $(\Gamma \vdash A \equiv B)$ via $M, M^{-1}$ iff

- $\forall P, \Gamma \vdash P : A \Rightarrow \Gamma \vdash (MP) : B$ and $\Gamma \vdash M^{-1}(MP) = P : A$
\(\forall T \to \text{no problem and it is easy to check that, given } N\)
\[
\text{Since these are closed terms, the context } \Gamma \text{ poses no problem and it is easy to check that, given } N\]

\(\text{peculiar typing rule used to obtain the traditional polymorphism in ML-style languages.}\)

\(\text{th is more than just \(T\)h also.}\)

\(\begin{align*}
\text{(VAR)} & \quad \Gamma \vdash x : A[\tau_i/X_i] \quad \text{if } x:A = \forall X_1 \ldots X_n.\tau \text{ is in } \Gamma \text{ and the } \tau_i \text{ are monotypes} \\
\text{(ABS)} & \quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x. M : A \to B} \\
\text{(PAIR)} & \quad \frac{\Gamma \vdash M : A_1 \quad \Gamma \vdash N : A_2}{\Gamma \vdash < M, N > : A_1 \times A_2} \\
\text{(LET)} & \quad \frac{\Gamma \vdash N : A \quad \Gamma, x : \forall X_1 \ldots X_n. A \vdash M : B}{\Gamma \vdash \langle x \rangle.MN : B}
\end{align*}\)

\(\begin{align*}
\text{(APP)} & \quad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \\
\text{(PROJ)} & \quad \frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash p_i M : A_i}
\end{align*}\)

\(\text{where } \{X_1 \ldots X_n\} \text{ is } FV(A) - FV(\Gamma)\)

\(\text{Table 3: Type inference rules for an ML-like functional language.}\)

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

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

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

\(\text{Now we can easily verify that Axiom 12 is in a sense still valid.}\)

\(\text{Remark 3.5 Let } A \text{ be } \forall X.\sigma, \text{ where } \sigma \text{ is isomorphic to } T \text{ via } M, M^{-1}. \text{ Then it is easy to check that } M, M^{-1} \text{ provide an ML-isomorphism between } \forall X.\sigma \text{ and } T \text{ also.}\)

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

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

\(\text{But the real surprise is that we also get a new isomorphism, not derivable in } Th^{2}_{\Sigma,T}, \text{ that comes out of the peculiar typing rule used to obtain the traditional let polymorphism in ML-style languages.}\)

\(\text{Proposition 3.6 In ML-like languages, the following isomorphism hold}\)

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

\(\text{Proof. It suffices to provide } M \text{ and } M^{-1} \text{ s.t. } \forall \Gamma, \Gamma \vdash A \cong B \text{ via } M, M^{-1}.\)

\(\text{Let } M \text{ be } \lambda x. < p_1 x, p_2 x > \text{ and } M^{-1} \text{ be } \lambda x x. \text{ Since these are closed terms, the context } \Gamma \text{ poses no problem and it is easy to check that, given } N\)

\(\text{s.t. } \Gamma \vdash N : \forall X.A \times B, \text{ we can derive, using as a key tool the let polymorphic type inference rule, that } \Gamma \vdash \langle x \rangle.(\lambda x. < p_1 x, p_2 x >)N : \forall X.\forall Y.A \times (B[Y/X]). \text{ Furthermore, it is clear that } (\lambda x x)(\langle x \rangle.(\lambda x. < p_1 x, p_2 x >)N) \text{ can be assigned type } \forall X.A \times B.\)

\(\text{The other direction of the isomorphism is obvious, since } \forall X.A \times B \text{ is an instance of } \forall X.\forall Y.A \times (B[Y/X]).\)

\(\text{Well, if you really don’t believe it, just run your favorite typed functional language and try the following example (syntax of CAML):}\)

\(\text{Example 3.7 CAML (mips) (V 2-6.1) by INRIA Fri Nov 24 1989}\)

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

\(\text{□}\)

\(\text{Remark 3.8 The isomorphism (split) is not derivable in } Th^{2}_{\Sigma,T}.\)

\(\text{Indeed, (split) allows to change the number of free type variables even in types that are not isomorphic to the unit type } T, \text{ while all the axioms in } Th^{2}_{\Sigma,T} \text{ preserve that number for such types. This fact is particularly unexpected, as it shows that type-assignment systems allow to prove constructively equivalent some proofs that are not so in the second order logic corresponding to the second order } \lambda\text{-calculus (for a discussion of the notion of constructive equivalence, and its connections with the isomorphisms of types, see [DCL89]). So the original commonplace idea that } ML \text{ is just a limited version of second order } \lambda\text{-calculus is now deeply shaken: in (core) } ML \text{ we cannot do everything we can do in explicitly polymorphic calculi, as it}\)
is well known, but it is also surprisingly true that we can do in (core) ML something that cannot be done in second order \( \lambda \)-calculus.

4 Completeness and conservativity results

Are there any more unexpected isomorphisms coming out of the let construct? What about the Axioms 10 and 11 of \( \text{Th}^2_{\lambda \land T} \) we were forced to leave out? Do they induce some derived isomorphisms on ML types? These are the questions we address in the present Section.

4.1 Completeness

By adapting to the type assignment framework the techniques introduced in [BDCL90] and [DC91], we can prove the following fundamental result.

Theorem 4.1 The theory \( \text{Th}^2_{\lambda \land T} \) less Axioms 10, 11 and 12 plus (unit) and (split) is complete for ML isomorphisms.

Proof. See Appendix. \( \square \)

This result gives us the safe definition of the theory \( \text{Th}^{ML} \) of type isomorphisms for (core) ML:

Definition 4.2 \( \text{Th}^{ML} \) is the theory of equality defined by \( \text{Th}^2_{\lambda \land T} \) less Axioms 10, 11 and 12 plus (unit) and (split).

4.2 Conservativity

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

Proposition 4.3 Let \( A \) and \( B \) be ML types. If \( \text{Th}^2_{\lambda \land T} \) proves \( A = B \), then \( \text{Th}^{ML} \) proves \( A = B \) too.

Proof. See Appendix. \( \square \)

Since (split) is not derivable in \( \text{Th}^2_{\lambda \land T} \) (Remark 3.8), the theory \( \text{Th}^{ML} \) is strictly more powerful on ML types, so the previous proposition actually states that \( \text{Th}^{ML} \) is an extension of \( \text{Th}^2_{\lambda \land T} \) on ML types, and not the reverse.

4.3 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 rewritten to a (unique) type normal form n.f.(\( A \)) via a strongly normalizing confluent\(^1\) type

rewriting system derived from the axioms of \( \text{Th}^{ML} \).

Definition 4.4 (Type rewriting \( R \)) Let \( \sim \) be the transitive and substitutive type-reduction relation generated by:

\[
A \times (B \times C) \sim (A \times B) \times C \quad T \times A \sim A \\
(A \times B) \rightarrow C \sim A \rightarrow (B \rightarrow C) \quad A \rightarrow T \sim T \\
A \rightarrow (B \times C) \sim (A \rightarrow B) \times (A \rightarrow C) \quad T \rightarrow A \sim A \\
A \times T \sim A \quad \forall X. T \sim T.
\]

Remark 4.5 A type normal form n.f.(\( A \)) of a type \( A \) is just a type \( \forall X_1 \ldots X_n. (A_1 \times \ldots \times A_n) \), where no product or unit type appear in the \( A_i \). We call the \( A_i \) the coordinates of \( A \).

It can be shown that \( \text{Th}^{ML} \) proves \( A = B \) iff n.f.(\( A \)) is proved equal to n.f.(\( B \)) via (split), associativity and commutativity of product, bound variable renaming, quantifier swap and the derived Axiom (swap).

To decide this last equality, we can use (split) to rename all the bound variables in such a way that in the normal forms the \( A_i \) share no common type variable. We will call split-normal-form a type normal form with this property.

Using again the analysis of the structure of the terms that witness the isomorphism used in the proof of Theorem 4.1, it is then easily shown that \( \text{Th}^{ML} \) proves \( A = B \) iff the coordinates of the split-n.f. of \( A \) and \( B \) are in the same number and for a permutation \( \sigma \) each \( A_i \) is equal to some \( B_{\sigma(i)} \) via variable renaming, and (swap).

Since unification up to (swap), which is the left-commutativity of \( \rightarrow \), is decidable (see [Kir85]), this last problem is easily solved by looking for a variable renaming unifier that does not identifies variables originally distinguished inside split-n.f.(\( A \)) or split-n.f.(\( B \)).

A detailed account of the decision procedure will be given in [DC92].

5 Understanding ML polymorphism: completing the type checker

Actually, there is something special in (split) 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 \( \lambda x. p_i x \) and \( \lambda x. < p_i x, p_2 x > \).

Indeed, the only interesting effect of the term \( \lambda x. < p_i x, p_2 x > \) 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-checking 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. Now, it is doubtful if the isomorphisms in $T^{=}_{\lambda T}$ 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 $T^{=}_{\lambda T}$ 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 case (essentially by ruling out commutativity), but the matter is not clear enough to suggest such a modification right now.

- **Complexity:** unification up to $T^{=}_{\lambda T}$ is not known to be decidable (see [NPS89] for recent results), and even equality up to $T^{ML}_{\lambda T}$ is at least as hard as Graph Isomorphism (see [DC91] and [Bas90] for details), so such a modification of the ML type-checker is not clearly feasible.

But these problems are not there if we consider (split) alone: for correctness, there is nothing to prove, as 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 5.1
CAML (mips) (V 2-6.1) by INRIA Fri Nov 24 1989

```plaintext
#let join = let pair x = (x,x) in let id x = x in id in pair in id 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;;
```

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

```plaintext
let x = e' in e ⇒ (λx.e) e'.
```

- join translates to

  $$(\lambda pair.(\lambda f.pair f)(\lambda x.x))(\lambda x.\langle x, x \rangle)$$

- split translates to $(\lambda f.\langle f, f \rangle)(\lambda x.x)$

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 $\beta$ conversion. Actually, join $\beta$-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 $(\lambda x.M)N$, we change the type-inference algorithm, that would try to give a type to $(\lambda x.M)$ 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 $(\lambda x.M)N$ 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 to moving forward more than one step in the type-inference process.

Remark 5.2 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, $(\lambda x.\lambda y.y)\Omega$, where $\Omega$ is a diverging term, is of course not typable, while its reduct $\lambda y.y$ 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.

5.1 A modified type inference algorithm featuring 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 4, 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 5.1 above.

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.

6 Conclusions

As the discovery of the new isomorphism (split) stresses, it is not possible to consider ML-style languages as a particular case of the explicitly typed languages: this paper provides, as far as we know, the first explicit treatment of isomorphic types in the framework of type-assignment systems.

The main contributions of this work are the characterization of the class of isomorphic types and the extension of the ML type checker. The first result provides the necessary theoretical basis for the design of tools to perform library searches using the type of a function as a search key. Previous work on the subject originally motivated this research, and finds here its natural completion.

The extension to the ML type checker derived from the (split) isomorphism rises on the contrary some new issues. The traditional way of typing let expressions corresponds to typing programs that will be typable without let after one step of reduction. The new rule to capture (split) seems to correspond to moving forward two steps in the reduction: we believe that the necessity of moving two step forward is related in an essential way to the non linearity of the Surjective Pairing rule, that is the counterpart of η-equality in the theory of ML with products. It is probably for this reason that the ML type checker, originally born without tuple constructors, failed to incorporate a rule similar to (split) from the beginning. This is why, as suggested in the title of the paper, the new rule (split-let) is to be seen more as a completion of the original type inference system than as an extension to it.

We believe that it is necessary to understand more thoroughly than we do now the real nature of ML polymorphism, especially in the presence of type constructors different from the arrow: the case of the product we treated here tells us that we can be in for some more surprises. This investigation will be the argument of forthcoming work.

Acknowledgements

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.

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 $\lambda^2 \beta \eta \pi*$.

Thanks finally to Pierre Cregut and Delia Kesner for several afternoons spent discussing all these matters.

A Technical proofs.

This Appendix is meant to provide a sketch of the proofs of Theorems 4.1 and Proposition 4.3, and it is mainly here with the aim to give a taste of the proof techniques that were developed, not to provide the full details. The interested reader ought to refer to [DC91] and [DC92].
In particular, we will use in what follows many notion whose definition can be best found in the references. For the notion of finite-hereditary-permutations (f.h.p.’s) and Bohm tree (BT(M)) of a term M, see [Bar84], [BDCL90]. For second order finite-hereditary-permutations (2-f.h.p.’s), see [BL85] and especially [DC91].

A.1 Completeness

To show completeness of $\text{Th}^\text{ML}$, we first notice that each type reduction rule in $\rightarrow$ (see Definition 4.4) 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.

$$
\begin{aligned}
A & \leftarrow F \quad \forall \bar{X}.(A_1 \times \ldots \times A_n) \\
M & \quad M' = G \circ M \circ F^{-1} \\
B & \leftarrow G \quad \forall \bar{Y}.(B_1 \times \ldots \times B_m)
\end{aligned}
$$

Figure 1: Decomposition of an ML isomorphism.

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 $\text{Th}^\text{ML}$, 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] for the case of explicitly typed languages, that we adapt here to the type assignment framework.

Since to deal with the structure of terms we need to work on normal form representatives of terms, 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.1 The reduction rule for Surjective Pairing

(SP) $\langle p_1M, p_2M \rangle$ reduces to $M$

strictly decreases the set of types that can be assigned to a term by the type-inference algorithm of Definition 3.1.

Indeed, the program split in Example 5.1 has the type $(('a -> 'a)*('b -> 'b))$, but its reductum w.r.t. (SP) can only have types that are instances of $(('a -> 'a)*('a -> 'a))$.

If we orient (SP) the other way round, though, to get a Surjective Pairing Expansion as suggested for example in [Jay91], it is easy to show that we still get a strongly normalizing calculus for which the reductum of a term $M$ can be given at least all the types that are legal for $M$.

Theorem A.2 (Subject reduction) Let $M$ reduce to $M'$ w.r.t. the usual notion of reduction, but with SP Expansion. If $M \vdash A$, then $M' \vdash A$.

Proof. Essentially the same as in [HS80], Theorem 15.17. □

Now we can carry on our analysis of invertible terms. Lemma 2.6 and Proposition 3.4 in [BDCL90] go through essentially unchanged in the type assignment case, and they tell us that isomorphic types in split-normal-forms have the same number of coordinates, so that, in Figure 1, $n = m$. Furthermore, for any given isomorphisms $M$ 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 $B_{\sigma(i)}$. Such $M_i$ are then finite-hereditary-permutations, whose structure is known from [Dez76], and the following Completeness Theorem can be shown by induction on the depth of the Bohm tree of $M$, exactly as in [BDCL90].

Theorem A.3 The theory $\text{Th}^2_T$ less Axioms 10, 11 and 12 plus (unit) and (split) is complete for ML isomorphisms.

Proof. Proceed as in [BDCL90], Theorem 3.5, with Axiom 8 and 9 on top of Axiom (swap) to take care of the additional cases arising from type assignment. For example, let’s do the base case.

- depth(BT(M)) = 1. Then $M$ is $\lambda x.x$, and can prove the isomorphisms $A \cong A_i$ for any type, or, for any renaming $\sigma$, $\forall \bar{X}. A \cong \forall \bar{Y}. A[\sigma(i)/X_i]$, due to the fact that in ML types the order and the names of the generic type variables are not relevant. In any case, $\text{Th}^\text{ML}$ proves these equalities: the first one trivially as $\text{Th}^\text{ML}$ is a theory of equality; the second one by Axioms 8 and 9.

□
A.2 Conservativity

Lemma A.4 Let $M : A \rightarrow B$ be a 2-f.h.p. (in normal form). If $A$ and $B$ are types not containing quantifiers, then $M$ is a term of $\lambda^3 \beta \eta$ (the simple typed $\lambda$-calculus) and Axiom (swap) suffices to prove $A = B$.


Theorem A.5 Let $\forall \vec{X}.A$ and $\forall \vec{Y}.B$ be second order types such that $A$ and $B$ do not contain quantifiers, products and the unit type. If $Th^2_{xT} \vdash \forall \vec{X}.A = \forall \vec{Y}.B$, then $Th^{ML} \vdash \forall \vec{X}.A = \forall \vec{Y}.B$.

Proof. Suppose that the given types are equal in $Th^2_{xT}$. 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, $Th^2_{xT}$ 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 $\vec{X}$ is equal to that of $\vec{Y}$.

Knowing all this, let’s study the term $M$. It is a 2-f.h.p., so (see [DC91], Definition 3.29)

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

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 $\vec{Y}$ in some order. This means that, for the permutation $\sigma : n + k \rightarrow n + k$ associated to the 2-f.h.p. $M$, we have that $\lambda x_i.\sigma_P(i)$ 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

$$M = \lambda z : (\forall \vec{X}.A).\lambda Y_1 \ldots Y_n.\lambda x_{n+1} \ldots x_{n+k}.z[\sigma(Y_1) \ldots \sigma(Y_n)]P_{n+1} \ldots P_{n+k},$$

where we know by Lemma A.4, that the 2-f.h.p.’s $\lambda x_i.\sigma_P(i)$ (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 $\forall \vec{X}.A = \forall \vec{Y}.B$ can be proved using only (swap) and Axioms 8 and 9, that are all derivable in $Th^{ML}$. □

Corollary A.6 Let $\forall \vec{X}.A$ and $\forall \vec{Y}.B$ be second order types as above in Theorem A.5. Let $\forall \vec{X}'.A$ and $\forall \vec{Y}'.B$ be the ML types obtained from them by erasing all quantifications on type variables not occurring in $A$ and $B$ respectively. Then $Th^2_{xT} \vdash \forall \vec{X}.A = \forall \vec{Y}.B \Rightarrow Th^{ML} \vdash \forall \vec{X}'.A = \forall \vec{Y}'.B$.

Proof. Suppose $Th^2_{xT} \vdash \forall \vec{X}.A = \forall \vec{Y}.B$.

The terms $P_{n+i}$’s and the variables $x_i$’s in Theorem A.5 contain as free type variables only the $Y_i$’s, as only these variables occur in the type $B$, so we can build the term

$$M' = \lambda w : (\forall \vec{X}'.A).\lambda \vec{Y}'.\lambda x_{n+1} \ldots x_{n+k}. w[\sigma(Y_1)P_{n+1} \ldots P_{n+k}]$$

Where $Y_i'$ is what is left of $Y_{\sigma(i)}$ after erasing the type variables not occurring in $B$.

The term $M'$ type checks, and proves (in $Th^2_{xT}$)

$$\forall \vec{X}'.A = \forall \vec{Y}'.B,$$

so we can apply once more Theorem A.5 and finally get $Th^{ML} \vdash \forall \vec{X}'.A = \forall \vec{Y}'.B$, as required. □

Theorem A.7 ($Th^{ML}$ subsumes $Th^2_{xT}$, on ML types)

Let $C$ and $D$ be any ML types. If $Th^2_{xT} \vdash C = D$, then $Th^{ML} \vdash C = D$.

Proof. Let $C = \forall \vec{X}.A$ and $C = \forall \vec{Y}.B$ be ML types equated in $Th^2_{xT}$. 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 $Th^2_{xT}$, 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 $\rightarrow \times$, 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 $C_i \equiv \forall \vec{X}.A_i$ and $D_i \equiv \forall \vec{Y}.B_i$. Now, Theorem 3.32 in [DC91] says that there exist a permutation $\sigma : n \rightarrow n$ s.t. for all $i Th^2_{xT} \vdash \forall \vec{X}.A_i = \forall \vec{Y}.B_i$. Let’s call $\vec{X}_i$ and $\vec{Y}_i$ the type variables free in the $A_i$’s and the $B_i$’s respectively. Now Corollary A.6 states that $Th^{ML} \vdash \forall \vec{X}_i.A_i = \forall \vec{Y}_i.B_i$ (since we can rename bound type variables in $Th^{ML}$, these equalities can be turned into $Th^{ML} \vdash \forall \vec{X}'.A' = \forall \vec{Y}'.B'$ where all the type variables have been renamed in such a way that no two $A_i$’s or $B_i$’s have the same type variable. If $M_i$’s are the ML terms associated to these equalities in $Th^{ML}$, then we can build the ML term

$$\lambda w.\langle M'_1(\sigma_P(1)w), \ldots, M'_n(\sigma_P(n)w) \rangle \ldots$$

that proves

$$Th^{ML} \vdash \forall \vec{X}_1' \ldots \vec{X}_n'.(A_1' \times \ldots \times A_n') = \forall \vec{Y}_1' \ldots \vec{Y}_n'.(B_1' \times \ldots \times B_n'$$

These two last types are in normal form w.r.t the type rewriting system $\sim$, that is a subsystem of $R$ in

\footnote{Notice that $\forall \vec{Y}'$ and $\vec{X}'$ have the same length, since the rules in $Th^2_{xT}$ do not change the number of bound variables to prove $\forall \vec{X}.A = \forall \vec{Y}.B$}
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.8 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.9
Let A and B be different types. Then it is easily seen that

$$Th^2_{X,T} \vdash \forall XY.(X \rightarrow (X \rightarrow Y) \rightarrow A) \times (Y \rightarrow (Y \rightarrow X) \rightarrow B)$$

$$= \forall ZW.(Z \rightarrow (Z \rightarrow W) \rightarrow B) \times (Z \rightarrow (Z \rightarrow W) \rightarrow A).$$

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

References


