Type isomorphisms in a type-assigment framework

From library searches using types to the completion of the ML type checker

Roberto Di Cosmo

LIENS (CNRS) - DMI
Ecole Normale Supérieure
45, Rue d'Ulm
75005 Paris - France
E-mail: roberto@dicosmo.org

Dipartimento di Informatica
Corso Italia, 40
56100 Pisa -Italy
E-mail: dicosmo@dipisa.DI.UNIPI.IT

Abstract

This paper contains a full treatment of isomorphic types for languages equipped with an ML style polymorphic type inference mechanism. Surprisingly enough, the results obtained contradict 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 not only 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, 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 isomorphisms 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 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 [Sol83]. 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 [BDCL90], where the relations between these settings and Proof Theory, originally suggested by Mints, have been studied, 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 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

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	foldl	$\forall X.\forall Y.(X \to Y \to X) \to X \to List(Y) \to X$
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_left	$\forall X.\forall Y.(X \times Y \to Y) \to Y \to List(X) \to Y$

Table 1: an example

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, foldl, fold* and *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 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 - 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).

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

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

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 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 λ-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 λ-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

$$\left.\begin{array}{l}
\left.\begin{array}{ll}
1. & A \times B = B \times A \\
2. & A \times (B \times C) = (A \times B) \times C \\
3. & (A \times B) \to C = A \to (B \to C) \\
4. & A \to (B \times C) = (A \to B) \times (A \to C) \\
5. & A \times \mathbf{T} = A \\
6. & A \to \mathbf{T} = \mathbf{T} \\
7. & \mathbf{T} \to A = A
\end{array}\right\} Th^1_{\times\mathbf{T}} \\
\\
\left.\begin{array}{ll}
8. & \forall X.\forall Y.A = \forall Y.\forall X.A \\
9. & \forall X.A = \forall Y.A[Y/X] \quad \textit{(X free for Y in A, Y not free in A)} \\
10. & \forall X.(A \to B) = A \to \forall X.B \quad \textit{(X not free in A)}
\end{array}\right\} Th^2 \\
\\
\begin{array}{ll}
11. & \forall X.A \times B = \forall X.A \times \forall X.B \\
12. & \forall X.\mathbf{T} = \mathbf{T}
\end{array}
\end{array}\right\} Th^2_{\times\mathbf{T}}$$

A, B, C can be arbitrary types and $\mathbf{T}$ 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

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

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

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 2.1 (Definable isomorphisms)
$A \cong B \iff$ there exist λ-terms $M : A \to B$ and $N : B \to A$ 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 $M : A \to B$ and $N : B \to A$.

Later on, this approach was extended to the lambda calculus with surjective pairing and terminal object ($\lambda^1\beta\eta\pi*$), 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 $Th^1_{\times\mathbf{T}}$ of Table 2.

2.2 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 $Th^2_{\times\mathbf{T}} \vdash$ A = 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 as far as we can go by now.

3 Isomorphisms of types in ML-style languages

In [Rit89] and [Rit90a], Rittri uses the theory $Th^1_{\times\mathbf{T}}$ 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*) (denoted by τ below) or the so-called type-schemas (denoted by σ 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 σ)*
monotypes	$\tau ::= At \mid X \mid \tau \to \tau \mid \tau \times \tau$

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 $\to$, $\times$ 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-schemas 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-schemas, it seems reasonable that Th^{ML} be just $Th^2_{\times\mathbf{T}}$ less these three axioms. So the naive approach to deciding equality of type-schemas $\sigma_1 = \forall\vec{X}.\tau_1$ and $\sigma_2 = \forall\vec{Y}.\tau_2$, would be to check if there is a way of substituting in some order the variables $\vec{X}$ with $\vec{Y}$ in τ_1 such that for the resulting type τ_1' the theory $Th^1_{\times\mathbf{T}}$ proves $\tau_1' = \tau_2$. We say naive, because 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 [Rit89]), 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-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* (λx.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 Γ ($\Gamma \vdash A \cong B$) via M, M^{-1} iff*

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

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

$$(ABS) \quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad (APP) \quad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash (MN) : B}$$

$$(PAIR) \quad \frac{\Gamma \vdash M : A_1 \quad \Gamma \vdash N : A_2}{\Gamma \vdash < M, N >: A_1 \times A_2} \qquad (PROJ) \quad \frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash p_i M : A_i}$$

$$(LET) \quad \frac{\Gamma \vdash N : A \quad \Gamma, x : \forall X_1 \ldots X_n.A \vdash M : B}{\Gamma \vdash (\lambda x.M)N : B} \qquad \textit{where } \{X_1 \ldots X_n\} \textit{ is } FV(A) - FV(\Gamma)$$

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

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

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, $M \circ M^{-1}$ = λx.x and vice-versa, so it is not necessary to require this property explicitly.

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

Remark 3.5 *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.

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

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

Proposition 3.6 *In ML-like languages, the following isomorphism hold*

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

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 $\lambda x. < \mathrm{p}_1 x, \mathrm{p}_2 x >$ and M^{-1} be λx.x. Since these are closed terms, the context Γ poses no problem and it is easy to check that, given N s.t. $\Gamma \vdash N : \forall X.A \times B$, we can derive, using as a key tool the let polymorphic type inference rule, that $\Gamma \vdash (\lambda x. < \mathrm{p}_1 x, \mathrm{p}_2 x >)N : \forall X.\forall Y.A \times (B[Y/X])$. Furthermore, it is clear that $(\lambda x.x)((\lambda x. < \mathrm{p}_1 x, \mathrm{p}_2 x >)N)$ can be assigned type $\forall X.A \times B$.

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

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

Example 3.7

```
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 f = join in (fst f, snd f);;
(<fun>,<fun>) : (('a -> 'a) * ('b -> 'b))
  □
```

Remark 3.8 *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 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 λ-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 is just a limited version of second order λ-calculus is now deeply shaken: in (core) ML 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 λ-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 $Th^2_{\times\mathbf{T}}$ 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 $Th^2_{\times\mathbf{T}}$ less Axioms 10, 11 and 12 plus* **(unit)** *and* **(split)** *is complete for ML isomorphisms.*

Proof. See Appendix. □

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

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

4.2 Conservativity

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 4.3 *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 Appendix. □

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

[1] The system $\rightsquigarrow$ is a sub-system of the one used in [DC91], see Proposition 3.5 there.

Definition 4.4 *(Type rewriting R) Let $\rightsquigarrow$ be the transitive and substitutive 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 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 Th^{ML} proves A = B iff n.f.(A) is proven 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 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 σ 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 $\to$, is decidable (see [Kir85]), this last problem is easily solved by looking for a *variable renaming* unifier that does not identifies variables originally distiguished 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 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-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 $Th^1_{\times\mathbf{T}}$ 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 case (essentially by ruling out commutativity), but the matter is not clear enough to suggest such a modification right now.

- **Complexity:** unification up to $Th^1_{\times\mathbf{T}}$ is not known to be decidable (see [NPS89] for recent results), and even equality up to Th^{ML} 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

#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 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 Ω 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

$$(\mathbf{split-let}) \quad \frac{\Gamma \vdash N : A \times B \quad \Gamma, x : \forall X_1 \dots X_n Y_1 \dots Y_m . A \times (B[Y_1 \dots Y_m / X_{i_1} \dots X_{i_m}]) \vdash M : C}{\Gamma \vdash (\lambda x.M) N : C}$$

The set $\{X_1 \dots X_n\}$ is $FV(A \times B) - FV(\Gamma)$,
$X_{i_1} \dots X_{i_m}$ are the type variables among these that are shared by A and B,
and $Y_1 \dots Y_m$ are fresh type variables.

Table 4: The rule **(split-let)**.

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 a 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 endebted 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 *Böhm 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 Th^{ML}, we first notice that each type reduction rule in $\leadsto$ (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*.

$$\begin{tikzcd}
\mathrm{A} \arrow[r, leftrightarrow, "\mathrm{F}"] \arrow[d, leftrightarrow, "\mathrm{M}"'] & \forall\vec{X}.(A_1 \times \ldots \times A_n) \arrow[d, leftrightarrow, dashed, "\mathrm{M'} = \mathrm{G}\circ\mathrm{M}\circ F^{-1}"] \\
\mathrm{B} \arrow[r, leftrightarrow, "\mathrm{G}"'] & \forall\vec{Y}.(B_1 \times \ldots \times B_m)
\end{tikzcd}$$

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 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] for the case of explicitly typed languages, that we adapt here to the type assignment framework.

Since to deal with the strucure 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*

$$(SP) \qquad \langle p_1 M, p_2 M\rangle \text{ reduces to } M$$

strictly decreases the set of types that can be assigned to a term by the type-inferernce algorithm of Definition 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 Böhm tree of M, exactly as in [BDCL90].

Theorem A.3 *The theory $Th^2_{\times\mathbf{T}}$ 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 λx.x, and can prove the isomorphisms A ≅ A, for any type, or, for any renaming σ, $\forall\vec{X}.A \cong \forall\vec{Y}.A[Y_{\sigma(i)}/X_i]$, due to the fact that in ML types the order and the names of the generic type variables are not relevant. In any case, Th^{ML} proves these equalities: the first one trivially as Th^{ML} is a theory of equality; the second one by Axioms 8 and 9.

□

A.2 Conservativity

Lemma A.4 *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 an easy induction on the Böhm tree of M. See [DC92] for details. □

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 $Th^2_{\times\mathbf{T}} \vdash \forall\vec{X}.A = \forall\vec{Y}.B$, then $Th^{ML} \vdash \forall\vec{X}.A = \forall\vec{Y}.B$.*

Proof. Suppose that the given types are equal in $Th^2_{\times\mathbf{T}}$. 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 $\vec{X}$ 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)

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

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 $\vec{Y}$ 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

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

where we know by Lemma A.4, 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 $\forall\vec{X}.A = \forall\vec{Y}.B$ 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 $Th^2_{\times\mathbf{T}} \vdash \forall\vec{X}.A = \forall\vec{Y}.B \Rightarrow Th^{ML} \vdash \forall\vec{X'}.A = \forall\vec{Y'}.B$*

Proof. Suppose $Th^2_{\times\mathbf{T}} \vdash \forall\vec{X}.A = \forall\vec{Y}.B$.
The terms P_{n+i}'s and the variables x_i's in Theorem A.5 contain as free type variables only the $\vec{Y'}_i$'s, as only these variables occur in the type B, so we can build the term

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

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[2], and proves (in $Th^2_{\times\mathbf{T}}$) $\forall\vec{X'}.A = \forall\vec{Y'}.B$, 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 $Th^2_{\times\mathbf{T}}$ 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 $= \forall\vec{X}.A$, and C $= \forall\vec{Y}.B$ 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) $= (C_1 \times \dots \times C_n)$ and n.f.(D) $= (D_1 \times \dots \times D_n)$, 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 $\to$ 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 $C_i \equiv \forall\vec{X}.A_i$ and $D_i \equiv \forall\vec{Y}.B_i$. Now, Theorem 3.32 in [DC91] says that there exist a permutation $\sigma : n \to n$ s.t. for all i $Th^2_{\times\mathbf{T}} \vdash \forall\vec{X}.A_i = \forall\vec{Y}.B_{\sigma(i)}$. 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 $Th^{ML} \vdash \forall\vec{X_i}.A_i = \forall\vec{Y_{\sigma(i)}}.B_{\sigma(i)}$ Since we can rename bound type variables in Th^{ML}, these equalities can be turned into $Th^{ML} \vdash \forall\vec{X'}_i.A'_i = \forall\vec{Y'}_{\sigma(i)}.B'_{\sigma(i)}$ 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

$$\lambda w.\langle M'_1(\mathrm{p}_{\sigma(1)}w), \langle \dots, M'_n(\mathrm{p}_{\sigma(n)}w)\rangle \dots\rangle$$

that proves

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

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

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

[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.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

$$Th^2_{\times\mathbf{T}} \vdash \forall XY.(X\to(X\to Y)\to A)\times(Y\to(Y\to X)\to B)$$
$$= \quad \forall ZW.(Z\to(Z\to W)\to B)\times(Z\to(Z\to W)\to A).$$

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

References

[Bar84] Henk Barendregt. *The Lambda Calculus; Its syntax and Semantics (revised edition).* North Holland, 1984.

[Bas90] 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.

[BDCL90] 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 : MSCS.

[BL85] 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.

[CDC91] 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, *ICALP*, pages 291–302, Springer-Verlag, 1991.

[CH88] Guy Cousineau and Gerard Huet. *The CAML primer.* Technical Report, LIENS - Ecole Normale Supérieure, 1988.

[DC91] 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.

[DC92] Roberto Di Cosmo. *Deciding Type isomorphisms in a type assignment framework.* Technical Report, LIENS - Ecole Normale Supérieure, 1992. To appear.

[DCL89] 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.

[Dez76] Mariangiola Dezani-Ciancaglini. Characterization of normal forms possessing an inverse in the λβη calculus. *Theoretical Computer Science*, 2:323–337, 1976.

[Hin69] R. Hindley. The principal type-scheme of a an object in combinatory logic. *Transactions of the American Mathematical Society*, 146, 1969.

[HS80] Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-calculus.* London Mathematical Society, 1980.

[Jay91] C. Barry Jay. Strong normalisation for simply-typed lambda-calculus as in lambek-scott. February 1991. LFCS, University of Edimburgh.

[Kir85] Claude Kirchner. *Methodes et utiles de conception systematique d'algoritmes d'unification dans les theories equationnelles.* PhD thesis, Université de Nancy, 1985.

[Mar72] C.F. Martin. Axiomatic bases for equational theories of natural numbers. *Notices of the Am. Math. Soc.*, 19(7):778, 1972.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17(3):348–375, 1978.

[NPS89] Paliath Narendran, Frank Pfenning, and Rick Statman. On the unification problem for cartesian closed categories. *Hardware Verification Workshop*, September 1989.

[Rit89] Mikael Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1), 1989.

[Rit90a] 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.

[Rit90b] Mikael Rittri. *Searching program libraries by type and proving compiler correctness by bisimulation.* PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 1990.

[RT89] Colin Runcyman and Ian Toyn. Retrieving re-usable software components by polymorphic type. *Fourth Int. Conf. on Functional Programming Languages and Computer Architecture*, 1989.

[Sol83] Serjey V. Soloviev. The category of finite sets and cartesian closed categories. *Journal of Soviet Mathematics*, 22(3):1387–1400, 1983.