October 1990

Type Inference for Records in a Natural Extension of ML

Didier Rémy

Type Inference for Records in a Natural Extension of ML

Abstract

We describe an extension of ML with records where inheritance is given by ML generic polymorphism. All operations on records introduced by Wand in [Wan87] are supported, in particular the unrestricted extension of a field, and other operations such as renaming of fields are added. The solution relies on both an extension of ML, where the language of types is sorted and considered modulo equations [Rem9Ob], and on a record extension of types [Rem9Oc]. The solution is simple and modular and the type inference algorithm is efficient in practice.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-90-73.

Type Inference For Records In A Natural Extension Of ML

MS-CIS-90-73
LOGIC & COMPUTATION 24

Didier Rémy

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

October 1990

Type Inference for Records in a Natural Extension of ML

Didier Rémy[*]
Departement of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

remy@linc.cis.upenn.edu

Abstract

We describe an extension of ML with records where inheritance is given by ML generic polymorphism. All operations on records introduced by Wand in [Wan87] are supported, in particular the unrestricted extension of a field, and other operations such as renaming of fields are added. The solution relies on both an extension of ML, where the language of types is sorted and considered modulo equations [Rem90b], and on a record extension of types [Rem90c]. The solution is simple and modular and the type inference algorithm is efficient in practice.

Introduction

The aim of typechecking is to guarantee that well-typed programs will not produce runtime errors. A type error is usually due to a programmer's mistake, and thus typechecking also helps him in debugging his programs. But programmers do not like writing the types of their programs by hand. Type inference requires as little type information as the declaration of data structures; then all types of programs will be automatically computed.

Our goal is to provide type inference for labelled product data structures, commonly called records, allowing some inheritance between them.

After recalling related work and defining the operations on records, we first review the solution for a finite (and small) set of labels, which was presented in [Rem89], then extend it to a denumberable set of labels. In the last part we discuss the power and weakness of the solution, describe some variations, and suggest improvements.

Why records?

Before records, data structures were built using product types, as in ML for example.

```
("Peter", "John", "Professor", 27, 5467567, 56478356, ("toyota", "old", 8929901))
```

With records one would write, instead:

```
{name = "Peter"; lastname = "John"; job = "Professor"; age = 27; id = 5467567;
 license = 56478356; vehicle = {name = "toyota"; id = 8929901; age = "old"}}
```

*Partly supported by research grant NSF IRI86-10617.

The latter program is definitely more readable than the former. It is more precise, too. Records can also help send arguments to functions or retrieve their results. More generally, in communication between processes records permit the naming of the different ports on which processes can exchange information. One nice example of this can be found in the language LCS [Ber88] which is a combination of the language ML and the language CCS designed by Robin Milner in 1980 [Mil80]. But certainly records have become more popular since we know that they can help implement objects with a kind of inheritance [Wan89, CM89].

If both variants and records were added, types of data structures could be completely inferred without any type declaration, whereas ML requires concrete data type declarations. This is also a strong motivation for having records.

Related work

Luca Cardelli is the first who claimed that functional languages should have record operations. Because he did not know how to combine records and polymorphism, he designed Amber in 1986 as a monomorphic language. Later he designed the language FUN where bounded quantification was introduced. Bounded quantification is an extension of second order quantification when some type inclusions are allowed. This notion is essential in coding operations on records with inheritance. In the language QUEST, the successor of FUN [CW85], bounded quantification was extended to higher order.

A lot of work has been done on the semantics of languages with inclusion, initially without record operations, which where incorporated later. It is only recently that a semantics of Quest has been proposed [LC90].

A slight but significant improvement of bounded quantification has been made in [CCH*89] to better consider recursive objects; a more general but less tractable system was studied by Pavel Curtis [Cur87]. Today, the interest seems the simplification rather than the enrichment of existing systems [LC90]. An interesting study whose goal is to remove bounded quantification is [HP90].

Records have also been formulated with explicit labelled conjunctive types in the language Forsythe [Rey88].

In contrast, records in implicitly typed languages have been less studied and all proposed extensions of ML are still very restrictive. The language Amber in 1986 did not actually make records more popular in functional langages, probably because it was not polymorphic. Inheritance in Amber was obtained by type inclusion [Car84, Car86]. Records became very attractive in 1987, after Mitchell Wand proposed a system in [Wan87] where inheritance was obtained from ML generic polymorphism. Though type inference was incomplete for this system, it remains a reference, for it was the first concrete proposal for extending ML with records having inheritance. Next year complete type inference algorithms were found for a strong restriction of this system [JM88, OB88]. They only allowed the extension of a record with a field that was not defined before. Then, the present author proposed a complete solution to Wand's system [Rem89], but it was formalized only in the case of a finite set of labels (a solution was also given by Wand in 1988, but the completeness was obtained at the cost of a complete set of principal types and the algorithm was explosive in practice). Mitchell Wand revisited this approach and extended it with an "and" operation[1] but did not provide correctness proofs. The case of an infinite set of labels was addressed in [Rem90a], which we review in this article. Works have been contributed by Peter Buneman and Atsushi Ohori, simplifying the previous system [OB89, Oho90]. Though the solution of [Oho90] solves only the restricted extension, it shares with this work a reliance on a sorted extension of ML, and pushes some of the label constraints to the level of sorts.

[1]You may understand it as an "append" on association lists in lisp compared to the "with" operation which should be understood as a "cons".

Operations on records

In this subsection we show with examples what operations on records are expected and introduce the main constructions. We use a CAML like syntax [CH89, Wei89].

The example we started with already illustrates a few ideas about labels. Like variable names, labels do not have particular meanings. Though choosing good names (good is very subjective) will help in writing and reading programs. Names can, of course, be reused in different records, even to build fields of different types. This is illustrated in the three following examples

```
let car = {name = "Toyota", age = "old"; id = 7866};;

let truck = {name = "Blazer", id = 6587867567};;

let person = {name = "Tim"; age = 31; id = 5656787};;
```

Note that no declaration previous to the use of labels is needed. The record person is defined on exactly the same fields as the record car, though those fields do not have the same intuitive meaning. The field age holds values of different types in car and person.

In the previous examples we built records all at once. But we can also do it step by step. A value driver can be defined as being a copy of the record person but with one more field vehicle filled with the previously defined car object.

```
let driver = {person with vehicle = car};;
```

Note that there is no sharing between the records person and driver. You can simply think as if the former were discharged into a new empty record before adding a field car to build the latter. This construction is called the *extension* of a record with a new field. In this example the field newly defined was not present in the record person, but that should not be a restriction. For instance, if our driver needs a more robust vehicle, we write

```
let truck_driver = {driver with vehicle = truck};;
```

As above, the operation is not a physical replacement of the vehicle field by a new value[2]. We do not wish any constraint between the types of the old and the new values of the vehicle field. To distinguish between the two kinds of extensions of a record with a new field, we will say that the extension is *strict* when the new field could not be previously defined and *unrestricted* otherwise.

A more general operation than adding a field to a record is the construction of a new record from two previously defined ones, taking the union of their defined fields. For instance, assume that a car has a good engine but a rusty body and that you cannot start your truck. If you are a good mechanic, you could build this strange object

```
let repair_truck = {car and truck};;
```

and drive again. Of course, the semantics of the and construction has to be defined. One question which arises is what value should be assigned to fields which are in both car and truck? Usually when there is a conflict, i.e. a same field is defined in both records, its value would be taken from the last record. But you might also expect from a typechecker that it would prevent this situation from happening. Although the and construction is less common in the literature, probably because it causes more trouble, it seems a very interesting one in different respects. This is what happens in the language SML [HMM90] when a structure is opened and extended with another one. In the language LCS the visible ports of two processes run in parallel are exactly the ports visible in any of them. And as shown by Mitchell Wand [Wan89] multiple inheritance can be coded with an and construction.

The constructions we described above are not exhaustive but are the more common ones. We should also mention the permutation, renaming and erasure of fields. We described how to build

[2]This operation would be ill typed if truck and car had incompatible types.

records, but of course we also want to read them. There is actually a unique construction for that purpose.

```
let id x = x.id;;
```

```
let age x = x.age;;
```

This shows that the construction which reads any specified field is a real functional value. But as labels are not values, there is no function which could take a label and a record as arguments and would read the field of the record corresponding to that label. Thus we need one extraction function per label, as for id and age above. Then they can be applied to different records of different types but all having the field we want to read. For instance

```
age person, age driver;;
```

They can also be passed to other functions, as in

```
let car_info field = field car;;
```

```
car_info age;;
```

The testing function

```
let eq x y = equal x.id y.id
```

should of course accept arguments of different types provided they both have an id field of the same type.

```
eq car truck;;
```

These examples were very simple. We will answer them below, but we will also meet more tricky ones.

1 A small solution when the set of labels is finite

Though this solution will be made obsolete by the extension to a denumerable set of labels, we choose to present it first, since it is very simple and the extension will be based on the same ideas. It will also be a very decent solution in some cases when one wants only a few labels. And it will emphasize a method for getting more polymorphism in ML (in fact, we will not put more polymorphism in ML but we will make more use of it, sometimes in unexpected ways).

We will sketch the path from Wand's proposal to this solution, for it may be of some interest to describe the method which we think could be applied in other situations. As intuitions are rather subjective, and ours may not be yours, the section 1.1 can be skipped whenever it does not help.

1.1 The method

Records are partial functions from a set $\mathcal{L}$ of labels to the set of values. We simplify the problem by considering only three labels a, b and c. Records can be represented in three field boxes, once labels have been ordered:

a	b	c

Defining a record is the same as filling some of the fields with some values. For example, we will put the values 1 and *true* in the a and c fields and leave the b field undefined.

1		*true*

Typeckecking means forgetting some information about values, for instance we will identify two different numbers and only remember them as being numbers. The structure of types usually reflects the structure of values, but with fewer details. It is thus natural to type records values with partial functions from labels to types.

$$\mathcal{L} \longrightarrow \mathit{types}$$

We first make record types total functions on labels using an explicitly undefined constant *abs* ("absent").

$$\mathcal{L} \longrightarrow \mathit{types} \cup \{\mathit{abs}\}$$

In fact, we replace the union by the sum $\mathit{pre}\,(\mathit{types}) + \mathit{abs}$. We decompose record types as follows:

$$\mathcal{L} \longrightarrow [1, \mathit{Card}\,(\mathcal{L})] \longrightarrow \mathit{pre}\,(\mathit{types}) + \mathit{abs}$$

The first function is an ordering from $\mathcal{L}$ to the segment $[1, \mathit{Card}\,(\mathcal{L})]$ and can be set once and for all. Thus record types can be represented only by the second component, which is a tuple of length $\mathit{Card}\,(\mathcal{L})$ of values in $\mathit{pre}\,(\mathit{types}) + \mathit{abs}$. The previous example is typed by

1		*true*

$$\Pi(\ \mathit{pre}\,(\mathit{num})\ ,\quad \mathit{abs}\quad ,\ \mathit{pre}\,(\mathit{bool})\ )$$

A function $\mathit{extract}^a$ reading the a field shall accept as an argument any record having the a field defined with a value M, and return M. The a projection of the type of the argument must be $\mathit{pre}\,(t)$ if t is the type of M. We do not care whether other fields are defined or not, so their types may be anything, i.e. they are variables φ and ψ. The result has type α.

$$\mathit{extract}^a : \Pi(\mathit{pre}\,(\alpha), \varphi, \psi) \rightarrow \alpha$$

1.2 A formulation

Because we want a very restricted use of *pre* and *abs* symbols, the language of types will be a sorted free algebra, $\mathcal{F}(\Sigma, \mathcal{V})$. The set $\mathcal{C}$ of type symbols contains at least an arrow symbol $\rightarrow$ and two symbols *pre* and *abs*. We note ϱ the arity function. The set $\mathcal{K}$ is a pair of two sorts *usual* and *field* and the signature Σ of $\mathcal{C}$ is defined by:

$$\begin{array}{lrl} & \mathit{pre} : & \mathit{usual} \Rightarrow \mathit{field} \\ & \mathit{abs} : & \mathit{field} \\ & \Pi : & \underbrace{\mathit{field} \otimes \ldots \mathit{field}}_{\mathit{card}(\mathcal{L})} \Rightarrow \mathit{usual} \\ \forall f \in \mathcal{C} \setminus \{\mathit{pre}, \mathit{abs}, \Pi\}, & f : & \underbrace{\mathit{usual} \otimes \ldots \mathit{usual}}_{\varrho(f)} \Rightarrow \mathit{usual} \end{array}$$

The extension of ML with sorted types is straightforward. We will not formalize it further, since this will be subsumed in the next section. The inference rules are the same as in ML though the language of types is sorted. The typing relation defined by the inference rules is still decidable and admits principal solutions in the usual sense.

In this language, we may assume that the primitive environment is composed of the following assertions:

$$\begin{array}{rl} \mathit{null} : & \Pi\,(\mathit{abs}, \ldots \mathit{abs}) \\ \mathit{extract}^a : & \Pi\,(\varphi_1 \ldots, \mathit{pre}\,(\alpha) \ldots \varphi_l) \rightarrow \alpha \\ \mathit{new}^a : & \Pi\,(\varphi_1, \ldots \varphi_l) \rightarrow \alpha \rightarrow \Pi\,(\varphi_1 \ldots, \mathit{pre}\,(\alpha), \ldots \varphi_l) \end{array}$$

System Π_f

The *null* constant is the empty record. The $\mathit{extract}^a$ constant reads the a field from its argument, and the new^a constant extends its first argument on label a with its second argument.

2 Extension to large records

Though the previous solution is very simple, and perfect when there are only two or three labels involved, it is clearly no longer acceptable when the set of labels is getting larger. This is because record types are proportional to the size of this set even the type of the null record, which has no field defined. When a very local use of records is needed, the number of labels may be written with only one digit and the solution works perfectly. But in a large system where some records are used globally, the number of labels will quickly be over one hundred.

In any program the number of labels will always be finite, but with modular programming, the whole set of labels is not often known at the beginning (though in this case, some of the labels may be local to a module and solved independently). In practice, it is thus interesting to reason on an "open", i.e. denumerable, set of labels. From a theoretical point of view it is the only way of avoiding the meta-reasoning which would show that any computation done in a system with a small set of labels would still be valid in a system with a larger set of labels, and that the typing in the latter case could be deduced from the typing in the former case. The nice solution consists in working in a system where all potential labels are taken into account from the beginning.

In the first part we will illustrate the above discussion and describe the intuitions. Then we will formalize the solution in three steps. First we extend types with record types in a more general framework of sorted algebras; record types will be sorted types modulo equations. The next step describes an extension of ML where types are sorted taken modulo equations. Last, we apply the results to a special case, re-using the same encoding as for the finite case.

2.1 An intuitive approach

We first assume that there are only two labels a and b. Let r be the record equal to $\{a = 1;\ b = \mathit{true}\}$ and f the function which reads the a field. What happens when we apply f to r? Assuming f has type $t \to s$ and r has type r, we can apply f to r if the two types t and r are unifiable. In our example we have

$$\begin{aligned} &t : \Pi\,(a : \mathit{pre}\,(\alpha);\ b : \varphi_b), \\ &r : \Pi\,(a : \mathit{pre}\,(\mathit{num});\ b : \mathit{pre}\,(\mathit{bool})), \end{aligned}$$

and s is equal to α. The unification of t and r is done field by field, their most general unifier is

$$\begin{cases} \alpha \mapsto \mathit{num} \\ \varphi_b \mapsto \mathit{pre}\,(\mathit{bool}) \end{cases}$$

If we had one more label c, the types t and r would be

$$\begin{aligned} &t : \Pi\,(a : \mathit{pre}\,(\alpha);\ b : \varphi_b;\ c : \varphi_c), \\ &r : \Pi\,(a : \mathit{pre}\,(\mathit{num});\ b : \mathit{pre}\,(\mathit{bool});\ c : \mathit{abs}). \end{aligned}$$

and their most general unifier

$$\begin{cases} \alpha \mapsto \mathit{num} \\ \varphi_b \mapsto \mathit{pre}\,(\mathit{bool}) \\ \varphi_c \mapsto \mathit{abs} \end{cases}$$

We can again replay with one more label d. We would have the types

$$\begin{aligned} &t : \Pi\,(a : \mathit{pre}\,(\alpha);\ b : \varphi_b;\ c : \varphi_c;\ d : \varphi_d), \\ &r : \Pi\,(a : \mathit{pre}\,(\mathit{num});\ b : \mathit{pre}\,(\mathit{bool});\ c : \mathit{abs};\ d : \mathit{abs}). \end{aligned}$$

and their most general unifier

$$\begin{cases} \alpha \mapsto \mathit{num} \\ \varphi_b \mapsto \mathit{pre}\,(\mathit{bool}) \\ \varphi_c \mapsto \mathit{abs} \\ \varphi_d \mapsto \mathit{abs} \end{cases}$$

Since labels c and d do not appear neither in r nor in f, it was obvious that fields c and d would behave the same, and that all their type components would be equal up to renaming of variables, i.e. isomorphic types. So we can guess the component of the most general unifier on any new field ℓ only by taking a copy of its component on the c or d field. Instead of writing the types of all the fields, we need only to write a template for all fields whose types are isomorphic, and the types of significant fields, i.e. those which are not isomorphic to the template.

$$\begin{array}{l} t : \Pi\,(a : pre\,(\alpha)\,;\ b : \varphi_b\,;\ \infty : \varphi_\infty), \\ r : \Pi\,(a : pre\,(num)\,;\ b : pre\,(bool)\,;\ \infty : abs\,). \end{array}$$

The expression $\Pi\,((\ell : t_\ell)_{\ell \in I}\,;\ \infty : s_\infty)$ should be read as

$$\Pi\left(\ell : \begin{cases} t_\ell & \text{if } \ell \in I \\ s_\ell & \text{otherwise, where } s_\ell \text{ is a copy of } s_\infty \end{cases}\right)_{\ell \in \mathcal{L}}$$

But we can directly calculate the most general unifier without developing this expression, which will actually allow the set of labels to be infinite. We summarize the above different views in this figure:

Labels	a	b	c	d	∞
t	$pre\,(num)$	$pre\,(bool)$	abs	abs	abs
s	$pre\,(\alpha)$	φ_b	φ_c	φ_d	φ_∞
$t \equiv s$	$pre\,(num)$	$pre\,(bool)$	abs	abs	abs

This approach is so intuitive that it seems very simple. There is a difficulty though, due to the sharing among different templates. Sometimes a field has to be extracted from its template, because it must be unified with a significant field.

The macroscopic operation that we need is the transformation of a template t into a copy s which will be the type of the extracted field and another copy r to become the new template. We regenerate the template during an extraction mainly because of sharing. But it is also intuitive that once a field has been extracted the retained template should remember that and thus it cannot be the same. In order to keep sharing, we must extract a field step by step starting from the leaves.

For a template variable α, the extraction consists in replacing that variable by two fresh variables β and γ, more precisely by the term $\ell : \beta\,;\ \gamma$. This is exactly the substitution

$$\alpha \mapsto \ell : \beta\,;\ \gamma$$

For a small[3] term $f(\alpha)$, assume that we already extracted field ℓ from α, i.e. we have $f(\ell : \beta\,;\ \gamma)$, we now want to replace it by $\ell : f(\alpha)\,;\ f(\gamma)$. How can we do that? We simply ask it to be true, i.e. we assume the axiom

$$f(\ell : \beta\,;\ \gamma) \overset{!}{=} \ell : f(\alpha)\,;\ f(\gamma)$$

We do that for every symbol but Π[4]. We have built a record extension of types which is described in the next section.

2.2 Extending a sorted free algebra with record terms

Because we want a general solution where the encoding of records is not specified, we will study the problem independently of any particular signature and focus on the construction of record types.

[3]A term of height one.

[4]We could wish to do it for the symbol Π as well in order to allow the template to be composed of records itselft, but this will not be needed for our application and we will not study this complicated case here.

In this section we forget the semantic of types and simply think of them as the terms of a free sorted algebra.

Let $\mathcal{V}$ be a set of variables, $\mathcal{C}$ be a set of symbols and $\mathcal{K}$ be a set of sorts. We note $\mathcal{K}^+$ the union

$$\bigcup_{n>0} \mathcal{K}^n.$$

Let Σ be a function from $\mathcal{C}$ to $\mathcal{K}^+$, also called a signature of $\mathcal{C}$. We write $f :_\Sigma \iota_0 \otimes \ldots \iota_{n-1} \Rightarrow \iota_n$ for $\Sigma : f \mapsto (\iota_0, \ldots \iota_n)$. We extend the term of the the free sorted algebra $\mathcal{F}(\Sigma, \mathcal{V})$ with record terms.

Unsorted records terms

We call the terms of the free unsorted algebra $\mathcal{F}(\mathcal{D}, \mathcal{V})$ where $\mathcal{D}$ is the set of symbols

$$\mathcal{C} \cup \{\Pi\} \cup \{@_\ell \mid \ell \in \mathcal{L}\},$$

unsorted record terms . We write $a : \alpha\,;\ \beta$ for $\alpha @_a \beta$, and $a : \alpha\,;\ b : \beta\,;\ \gamma$ for $a : \alpha\,;\ (b : \beta\,;\ \gamma)$.

Example 1 The expressions

$$\Pi\,(a : \mathit{pre}\,(\mathit{num})\,;\ c : \mathit{pre}\,(\mathit{bool})\,;\ \mathit{abs}\,)$$

and

$$\Pi\,(a : \mathit{pre}\,(b : \mathit{num}\,;\ \mathit{num})\,;\ \mathit{abs}\,)$$

are unsorted record terms. In section 2.4 we will consider the former as a possible type for the record $\{a = 1\,;\ c = \mathit{true}\}$ but we will not give a meaning to the latter. Unsorted record terms are too many. We define record terms using sorts to constraint their formation. Only a few of the unsorted record terms will have associated record terms.

Record terms

Let $\mathcal{C}'$ be a subset of $\mathcal{C}$ (by default $\mathcal{C}'$ is taken to be $\mathcal{C}$), and $\mathcal{K}'$ a subset of $\mathcal{K} \times \mathcal{K}$ (by default $\mathcal{K}'$ is taken to $\mathcal{K} \times \mathcal{K}$).

Definition 1 *Record terms* are the terms of the free sorted algebra $\mathcal{F}(\Sigma' \times \Sigma'', \mathcal{V})$ where

- The set of sorts is the product $\mathcal{K} \times \bar{\mathcal{P}}$ where $\bar{\mathcal{P}}$ is composed of the set $\mathcal{P}$ of all the finite subsets of $\mathcal{L}$, extended with a sort constant ϵ.

$$\bar{\mathcal{P}} = \{\epsilon\} \cup \mathcal{P}_{\text{fin}}(\mathcal{L})$$

- the set of constants $\mathcal{D}'$ is

$$\{f^\epsilon \mid f \in \mathcal{C}\} \cup \{f^A \mid f \in \mathcal{C}', A \in \mathcal{P}\} \cup \{\Pi^{\iota \Rightarrow \kappa} \mid (\iota, \kappa) \in \mathcal{K}'\} \cup \{@_\ell^{\iota, A} \mid \iota \in \mathcal{K}, A \in \mathcal{P}, \ell \in \mathcal{L} \setminus A\}$$

  all symbols being distinct.

- the signature $\Sigma' \times \Sigma''$ is the product signature of Σ' and Σ'', i.e. for any symbol f in $\mathcal{D}'$

$$f :_{\Sigma' \times \Sigma''} (\iota_i, A_i)_{i \in [1,p]} \Rightarrow (\kappa, B) \iff f :_{\Sigma'} (\iota_i)_{i \in [1,p]} \Rightarrow \kappa \wedge f :_{\Sigma''} (A_i)_{i \in [1,p]} \Rightarrow B$$

  where the two signatures Σ' and Σ'' on $\mathcal{D}'$ are defined by the assertions

$$\begin{array}{lrr}
\forall f \in \mathcal{C}, & f^\epsilon :_{\Sigma'} \Sigma(f) & f^\epsilon :_{\Sigma''} \epsilon^{\varrho(f)} \Rightarrow \epsilon \\
\forall f \in \mathcal{C}',\ \forall A \in \mathcal{P}, & f^A :_{\Sigma'} \Sigma(f) & f^A :_{\Sigma''} A^{\varrho(f)} \Rightarrow A \\
\forall \iota \Rightarrow \kappa \in \mathcal{K}', & \Pi^{\iota \Rightarrow \kappa} :_{\Sigma'} \iota \Rightarrow \kappa & \Pi^{\iota \Rightarrow \kappa} :_{\Sigma''} \emptyset \Rightarrow \epsilon \\
\forall \iota \in \mathcal{K},\ \forall A \in \mathcal{P},\ \forall \ell \in \mathcal{L} \setminus A, & @_\ell^{\iota, A} :_{\Sigma'} \iota \otimes \iota \Rightarrow \iota & @_\ell^{\iota, A} :_{\Sigma''} \epsilon \otimes A \cup \{\ell\} \Rightarrow A
\end{array}$$

□

Superscript erasure

We define a function *erasure* from the record terms to the unsorted record terms which remove all the superscripts of symbols. It is easy to show that for any unsorted term t, any sort ι and any element A of $\bar{\mathcal{P}}$, there is at most one record term t' such that

1. t is the erasure of t',
2. (ι, A) is the signature of t'.

This property allows us to define a record term by giving its erasure and signature, which we shall usually do. Moreover we shall not write the signature when it is implicit in the context or does not matter.

Example 2 The erasure of

$$\Pi^{\mathit{field}\Rightarrow \mathit{usual}} \left(\mathit{pre}^{\emptyset}(\mathit{num}^{\emptyset})@_a^{\mathit{field},\emptyset} \left(\mathit{pre}^{\{a\}}(\mathit{bool}^{\{a\}})@_c^{\mathit{field},\{a\}} \mathit{abs}^{\{a,c\}}\right)\right)$$

is the unsorted record term

$$\Pi\,(a : \mathit{pre}\,(\mathit{num});\;\; c : \mathit{pre}\,(\mathit{bool});\;\; \mathit{abs}\,)$$

but there is no record term whose erasure would be

$$\Pi\,(a : \mathit{pre}\,(b : \mathit{num}\,;\;\; \mathit{num});\;\; \mathit{abs}\,)$$

Definition 2 The $(\mathcal{C}', \mathcal{K}')$-record extension of the free sorted algebra $\mathcal{F}(\Sigma, \mathcal{V})$ over the set of labels $\mathcal{L}$ is the sorted algebra $\mathcal{F}(\Sigma' \times \Sigma'', \mathcal{V})/E$ where E is a composed of

- distributivity axioms, for all symbols $f : (\iota_i)_{i\in[1,p]} \Rightarrow \kappa$ and finite subset of labels A which do not contain a,

$$f^A \left(\alpha_i @_a^{\iota_i, A} \beta_i\right)_{i\in[1,p]} \overset{!}{=} \left(f^{\epsilon}\,(\alpha_i)_{i\in[1,p]}\right) @_a^{\kappa, A} \left(f^{A\cup\{a\}}\,(\beta_i)_{i\in[1,p]}\right)$$

- left commutativity axioms for all sort ι and finite set of labels A which does contain labels a and b,

$$\alpha @_a^{\iota, A}(\beta @_b^{\iota, A\cup\{a\}} \gamma) \overset{!}{=} \beta @_b^{\iota, A}(\alpha @_a^{\iota, A\cup\{b\}} \gamma)$$

□

Without the superscrit, the equations are written

- distributivity

$$f\,(a : \alpha_i\,;\;\; \beta_i)_{i\in[1,p]} \overset{!}{=} (a : f\,(\alpha_i)_{i\in[1,p]}\,;\;\; f\,(\beta_i)_{i\in[1,p]})$$

- left commutativity

$$(a : \alpha;\;\; b : \beta;\;\; \gamma) \overset{!}{=} (b : \beta;\;\; a : \alpha;\;\; \gamma)$$

Example 3 In the term

$$\Pi\,(a : \mathit{pre}\,(\mathit{num});\;\; c : \mathit{pre}\,(\mathit{bool});\;\; \mathit{abs}\,)$$

we can replace abs by $b : \mathit{abs}\;;\;\; \mathit{abs}$ and use the left commutativity axioms to obtain the term

$$\Pi\,(a : \mathit{pre}\,(\mathit{num});\;\; b : \mathit{abs}\;;\;\; c : \mathit{pre}\,(\mathit{bool});\;\; \mathit{abs}\,)$$

In the term

$$\Pi\,(a\colon pre(\alpha)\,;\;\; \varphi)$$

we can substitute φ by $b\colon \psi_b\,;\;\; c\colon \psi_c\,;\;\; \psi$ to get

$$\Pi\,(a\colon pre(\alpha)\,;\;\; b\colon \psi_b\,;\;\; c\colon \psi_c\,;\;\; \psi)$$

which can be unified with the previous term field by field.

In [Rem90c] we proved the following theorem:

Theorem 1 *Unification in the record extension $\mathcal{F}(\Sigma' \times \Sigma'', \mathcal{V})/E$ of any free sorted algebra $\mathcal{F}(\Sigma, \mathcal{V})$ is decidable and unitary (every solvable unification problem has a principal unifier).*

2.3 Extending the types of ML with a sorted equational theory

In this section we consider a sorted regular theory $\mathcal{T}/E$ for which unification is decidable and unitary. Recall that a regular theory is one whose left and right hand sides of axioms always have the same set of variables. For any term t of $\mathcal{T}/E$ we write $\mathcal{V}(t)$ the set of its variables.

We studied the possibility of adding a sorted equational theory over the types of ML in [Rem90a]. We recall here the main definitions and results. The language ML that we study is pure lambda-calculus extended with a *LET* construction. We assume a set of variables V_E.

Definition 3 The set of terms in ML, is the smallest set containing V_E and such that if x is a variable and M and N are terms, then so are MN, $\lambda x.M$ and *let* $x = M$ *in* N. □

We define a relation $\vdash_{\mathcal{T}/E}$, or $\vdash$ for short, called *typing judgments* of the form $\Gamma \vdash M : \forall \mathcal{X} \cdot t$ where

- Γ is a typing environment, i.e. a list of assertions of the form $x : \forall \mathcal{X} \cdot s$ where x is an ML variable, $\mathcal{X}$ is a set of type variables and t a type,
- M is a term,
- $\mathcal{X}$ is a finite set of type variables and t is a type, and we abbreviate $\forall \emptyset \cdot t$ by t,
- the relation $\leq$ ("is more general than") is defined by

$$\forall \mathcal{X} \cdot t \leq \forall \mathcal{Y} \cdot s \iff \wedge \begin{cases} \exists \alpha : \mathcal{X} \to \mathcal{V},\ s =_E t\alpha \\ (\mathcal{V}(t) \setminus \mathcal{X}) \cap \mathcal{Y} = \emptyset \end{cases}$$

by the set of inference rules $(ML_{\mathcal{T}/E})$

$$\textit{(VAR)} \qquad \frac{\forall \mathcal{X} \cdot t = \Gamma(x)}{\Gamma \vdash x : \forall \mathcal{X} \cdot t}$$

$$\textit{(INST)} \qquad \frac{\Gamma \vdash M : \forall \mathcal{X} \cdot t \quad \forall \mathcal{X} \cdot t \leq \forall \mathcal{Y} \cdot s}{\Gamma \vdash M : \forall \mathcal{Y} \cdot s}$$

$$\textit{(GEN)} \qquad \frac{\Gamma \vdash M : \forall \mathcal{X} \cdot t \quad \mathcal{Y} \cap \mathcal{V}(\Gamma) = \emptyset}{\Gamma \vdash M : \forall \mathcal{X} \cup \mathcal{Y} \cdot t}$$

$$\textit{(FUN)} \qquad \frac{\Gamma[x : t] \vdash M : s}{\Gamma \vdash \lambda x.M : t \to s}$$

$$(APP) \qquad \frac{\Gamma \vdash M : t \to s \quad \Gamma \vdash N : t}{\Gamma \vdash M\ N : s}$$

$$(LET) \qquad \frac{\Gamma \vdash M : \forall \mathcal{X} \cdot t \quad \Gamma[x : \forall \mathcal{X} \cdot t] \vdash N : s}{\Gamma \vdash \text{let } x = M \text{ in } N : s}$$

$$(EQUAL) \qquad \frac{\Gamma \vdash M : t \quad t =_E s}{\Gamma \vdash M : s}$$

All but the *EQUAL* rules are the usual rules for ML. The *EQUAL* rule is necessary since the equality on types is modulo the equations.

The problem of type inference for ML is stated as follows. Given a context Γ and a term M, find all the substitutions α and type schemes $\forall \mathcal{X} \cdot t$ such that $\Gamma\alpha \vdash M : \forall \mathcal{X} \cdot t$. The principal type property means that the previous set is either empty which is decidable, or of the form $\{\alpha\beta, \forall \mathcal{X} \cdot t\beta \mid \beta \in \mathcal{S}\}$ where the pair $(\alpha, \forall \mathcal{X} \cdot t)$, called the principal solution, is computable, and $\mathcal{S}$ denotes the set of all substitutions.

Theorem 2 *If the sorted theory $\mathcal{T}/E$ is regular and its unification is decidable and unitary, then the relation $\vdash_{\mathcal{T}/E}$ has the principal type property.*

2.4 Record type reconstruction

In this section we apply the two preceding theorems to extend the types of ML with records types, then we introduce the operations on records following the finite case.

We start with a set of symbols $\mathcal{C}$ containing at least the arrow symbol $\to$, and two symbols *pre* and *abs*. The Π symbol will be provided by the record extension. Let $\mathcal{V}$ be a denumerable set of type variables. Let $\mathcal{K}$ be the set of two sorts *usual* and *field*, and Σ the following signature of symbols in $\mathcal{C}$ over $\mathcal{K}$:

$$\begin{array}{rl} \forall f \in \mathcal{C}, \qquad f & : \mathit{usual}^{\,e(f)} \Rightarrow \mathit{usual} \\ \mathit{pre} & : \mathit{usual} \Rightarrow \mathit{field} \\ \mathit{abs} & : \mathit{field} \end{array}$$

We write $\mathcal{T}(\Sigma)$ for the free algebra $\mathcal{F}(\Sigma, \mathcal{V})$ and $\mathcal{R}(\Sigma)/E$ the record extension $\mathcal{F}(\Sigma' \times \Sigma'', \mathcal{V})/E$ as defined in section 2.2.

We call ML(Σ) the language ML, with the typing relation $\vdash_{\mathcal{R}(\Sigma)/E}$. It follows from theorem 2 and the above properties that the relation $\vdash_{\mathcal{R}(\Sigma)/E}$ is decidable and has the principal type property.

Following the finite case we assume in the language ML(Σ) a primitive environment composed of the (denumerable) set of assertions

$$\begin{array}{rl} \mathit{null} & : \Pi\,(\mathit{abs}) \\ \mathit{extract}^a & : \Pi\,(a : \mathit{pre}(\alpha)\,;\ \varphi) \to \alpha \\ \mathit{new}^a & : \Pi\,(a : \varphi\,;\ \psi) \to \alpha \to \Pi\,(a : \mathit{pre}(\alpha)\,;\ \psi) \end{array}$$

System Π

It is convenient to use a smoother notation, adding the following macro-syntax facilities:

$$\begin{array}{rl} \{\} & \equiv \mathit{null} \\ \{r \text{ with } a = x\} & \equiv \mathit{new}^a\ r\ x \\ r.a & \equiv \mathit{extract}^a\ r \\ \{a_1 = x_1\,;\ \ldots a_n = x_n\} & \equiv \{\{a_1 = x_1\,;\ \ldots a_{n-1} = x_{n-1}\} \text{ with } a_n = x_n\} \end{array}$$

We illustrate this system by examples in the next section.

3 Playing with records

In this section we first show on very simple examples how most of the operations are solved by this system, then we meet its limitations. Some of them find a remedy by slightly improving the encoding. Last we propose and discuss some further extensions.

3.1 A demonstration of the game

A typeckecking prototype has been implemented in the language CAML. It was used to automatically type all the examples presented here and preceded by the # character. We start with very simple examples and end with a short program.

When building simple record values

```
#let car = {name = "Toyota"; age = "old"; id = 7866};;
car : {name : Pre (string); age : Pre (string); id : Pre (num); Abs}

#let truck = {name = "Blazer"; id = 6587867567};;
truck : {name : Pre (string); id : Pre (num); Abs}

#let person = {name = "Tim"; age = 31; id = 5656787};;
person : {name : Pre (string); age : Pre (num); id : Pre (num); Abs}
```

each field defined with a value of type t is significant with the type *pre*(t). Other fields are not, and are gathered in the template *abs*. We can extend a record with a new field,

```
#let driver = {person with vehicle = car};;
driver :
  {vehicle : Pre ({name : Pre (string); age : Pre (string); id : Pre (num); Abs});
   name : Pre (string); age : Pre (num); id : Pre (num); Abs}
```

whether the field was previously undefined as above, or defined as below:

```
#let truck_driver = {driver with vehicle = truck};;
truck_driver :
  {vehicle : Pre ({name : Pre (string); id : Pre (num); Abs}); name : Pre (string);
   age : Pre (num); id : Pre (num); Abs}
```

But we do not provide an and construction.

The only construction for accessing fields is the "dot" operation.

```
#let age x = x.age;;
age : {age : Pre ('a); 'p} → 'a

#let id x = x.id;;
id : {id : Pre ('a); 'p} → 'a
```

The accessed field must be defined with a value of typed 'a, so it has type pre ('a), and other fields may or may not be defined; they are gathered in the template= variable 'p. The return value has type 'a. To illustrate the plain functionality, we pass age as an argument to another function in the following example.

```
#let car_info field = field car;;
car_info : ({name : Pre (string); age : Pre (string); id : Pre (num); Abs} → 'a) → 'a

#car_info age;;
it : string
```

The function equal below takes two records both possessing an id field of the same type, but we do not care about the other fields. For simplicity of examples we assume a polymorphic equality equal on numbers.

```
#let eq x y = equal x.id y.id;;
eq : {id : Pre ('a); 'p} → {id : Pre ('a); 'q} → bool

#eq car truck;;
it : bool
```

We will see more examples in section 3.3. Let us turn to the counter-examples.

3.2 Limitations

There are mainly two kinds of limitations, one is due to the encoding method, the other to ML generic polymorphism.

The only source of polymorphism in record operations is generic polymorphism. Or we saw that in a record value, a field defined with a value of type t is typed by $pre\,(t)$. Once a field is defined every function must see it defined. This forbids merging two records with different sets of defined fields. We will use the following function to shorten examples

```
#let choice x y = if true then x else y;;
choice : 'a → 'a → 'a
```

We then fail with

```
#choice car truck;;
Typechecking error: collision between Pre (string) and Abs
```

because the age field is undefined in truck but defined in car. This is really a weakness, for the program

```
#(choice car truck).name;;
Typechecking error: collision between Pre (string) and Abs
```

which is equivalent (but more efficient) to the program

```
#choice car.name truck.name;;
it : string
```

may actually be useful. We will give a partial solution to this problem, and suggest a full but expensive one.

A natural generalization of the above eq function is to abstract the field which is used for testing equality

```
#let field_eq field x y = equal (field x) (field y);;
field_eq : ('a → 'b) → 'a → 'a → bool
```

It is so general that it could test equality of other values than records. We would get the old eq version by applying field_eq to the function id.

```
#let id_eq = field_eq id;;
id_eq : {id : Pre ('a); 'p} → {id : Pre ('a); 'p} → bool

#id_eq car truck;;
Typechecking error: collision between Pre (string) and Abs
```

The last example failed. This is not surprising since as field is bound by a lambda in field_eq, its two instances have the same type and so have both arguments x and y. Though in eq the arguments x and y were unlinked by two different instances of id. This is nothing else but ML generic polymorphism restriction. We emphasize that as record polymorphism is only generic the restriction applies drastically to them.

3.3 Flexibility and Remedies

The method for typeckecking records is very flexible. For the operation on records have not been fixed at the beginning, but at the very end. They are parameters which can vary in many ways.

The easiest thing to play with is changing the types of primitives. For instance asserting that new^a has the principal type

$$new^a : \Pi\,(a : abs\ ;\ \psi) \rightarrow \alpha \rightarrow \Pi\,(a : pre\,(\alpha);\ \psi)$$

will make the extension of a record with a new field possible only if the field was previously undefined. This slight change gives exactly the strong restriction that appears in both attempts to solve Wand's system [JM88, OB88]. Weakening the type of this primitive may be interesting in some cases, because the restricted construction may be easier to implement, and more efficient.

We can freely change the type of primitives, provided we will know how to implement them correctly, but we more generally can change the set of operations on records themselves. Since a defined field may not be dropped implicitly, it would be convenient to dispose of a primitive removing explicitly a field from a record

$$forget^a : \Pi\,(a : \varphi;\ \psi) \rightarrow \Pi\,(a : abs\ ;\ \varphi),$$

and add this syntactic facility

$$\{r \text{ without } a\} \equiv forget^a\ r.$$

Our encoding also allows us to type a function which renames fields

$$rename^{a \leftarrow b} : \Pi\,(a : \varphi;\ b : \psi;\ \chi) \rightarrow \Pi\,(a : abs\ ;\ b : \varphi;\ \chi)$$

Note that the renamed field may not be defined. In the result it is no longer accessible. A more primitive function would just exchange two fields

$$exchange^{a \leftrightarrow b} : \Pi\,(a : \varphi;\ b : \psi;\ \chi) \rightarrow \Pi\,(a : \psi;\ b : \varphi;\ \chi)$$

Then the *rename* constant is simply the composition

$$forget^a \circ exchange^{a \leftrightarrow b}.$$

But the flexibility is much more than that. The decidability of type inference does not depend of the specific signature of *pre* and *abs* type symbols. The encoding of records can be reviewed. And we are going to illustrate that by presenting another system for type-checking records.

We mentioned above that one extension of the current system should allow some polymorphism on records values themselves. We recall the example which we failed to type

```
#choice car truck;;
Typechecking error: collision between Pre (string) and Abs
```

because the age field was defined in car but undefined in truck. But we would like the result to have a type with *abs* on this field to guarantee than it will not be accessed, but common and compatible fields should remain accessible. The idea is that a defined field should be seen as undefined when needed. From the type point of view, this would require that a defined field with a value of type t should be typed with both $pre\,(t)$ and abs. If possible not using conjunctive types [Cop80].

The solution is first to force *abs* to be of arity 1 replacing each use of *abs* by $abs\,(\alpha)$ where α is a free, and so quantified, variable. Then because we cannot write

$$\forall \varphi \cdot \varphi(t)$$

where φ would range over *abs* and *pre*, we make *abs* and *pre* constant symbols by introducing an infix field symbol noted ".", and a new sort *flag*

$$\begin{aligned} pre &: flag \\ abs &: flag \\ . &: flag \otimes usual \Rightarrow field \end{aligned}$$

We now write $pre\,.\alpha$ instead of $pre\,(\alpha)$ and $\varepsilon.\alpha$ where ε is a variable of the sort *flag* . The solution Π^* is defined by the following set of primitives

$$\begin{aligned} null &: \Pi\,(abs\,.\alpha) \\ extract^a &: \Pi\,(a : pre\,.\alpha\,;\ \varphi) \to \alpha \\ new^a &: \Pi\,(a : \varphi\,;\ \psi) \to \alpha \to \Pi\,(a : \varepsilon.\alpha\,;\ \psi) \end{aligned}$$

System Π^*

It is easy to see that system Π^* is more general than system Π, i.e. that any expression typable in the system Π is also typable in the system Π^*: Replacing in a proof in Π all occurrences of *abs* by $abs\,.\alpha$ and all occurrence of $pre\,(t)$ by $pre\,.t$, where α does not appear in the proof, we obtain a correct proof in Π^*.

We retype some of the examples in the system Π^*. Building a record creates a polymorphic object, since all fields have a distinct *flag* variable

```
#let car = {name = "Toyota"; age = "old"; id = 7866};;
car : {name : 'u.string; age : 'v.string; id : 'w.num; abs.'a}
```

```
#let truck = {name = "Blazer"; id = 6587867567};;
truck : {name : 'u.string; id : 'v.num; abs.'a}
```

Now these two records can be merged,

```
#choice car truck;;
it : {name : 'u.string; age : abs.string; id : 'v.num; abs.'a}
```

forgetting the age field in car. Note that if the presence of field age has been forgotten, its type has not: we always remember the types of values which have stayed in fields. Thus we fail with

```
#let person = {name = "Tim"; age = 31; id = 5656787};;
person : {name : 'u.string; age : 'v.num; id : 'w.num; abs.'a}
```

```
#choice person car;;
Typechecking error: collision between num and string
```

This is really a failure since both records have common field name and id, which might be tested on later, and this example would be correct in the explicitly typed language QUEST [Car89]. If we add a new collection of primitives

$$forget^a : \Pi\,(a : \varphi\,;\ \psi) \to \Pi\,(a : \varepsilon.\alpha\,;\ \psi),$$

then we can turn around the above failure, explicitly forgetting label age in any of the two records

```
#choice {car without age} person;;
it : {name : 'u.string; age : abs.num; id : 'v.num; abs.'a}
```

```
#choice car {person without age};;
it : {name : 'u.string; age : abs.string; id : 'v.num; abs.'a}
```

```
#choice {car without age} {person without age};;
it : {age : abs.'a; name : 'u.string; id : 'v.num; abs.'b}
```

We can now present a more realistic example which illustrate the ability to add annotations on data structures and of course to type the presence of these annotations. The example is run into the system Π^*, and we assume an infix addition + of type num → num → num.

```
#type tree ('u) = Leaf of num
#              | Node of {left: pre.tree ('u); right: pre.tree ('u);
#                        annot: 'u.num; abs.unit}
#;;
New constructors declared:
Node : {left : pre.tree ('u); right : pre.tree ('u); annot : 'u.num; abs.unit} → tree ('u)
Leaf : num → tree ('u)
```

The variable 'u indicates the presence of the annotation annot. For instance this annotation is absent in the structure

```
#let winter = 'Node {left = 'Leaf 1; right = 'Leaf 2 };;
winter : tree (abs)
```

The following function annotates a structure.

```
#let rec annotation =
# function
#    Leaf n → 'Leaf n, n
#  | Node {left = r; right = s} →
#     let (r,p) = annotation r in
#     let (s,q) = annotation s in
#        'Node {left = r; right = s; annot = p+q}, p+q;;
annotation : tree ('u) → tree ('v) * num
```

```
#let annotate x = match annotation x with y,_ → y;;
annotate : tree ('u) → tree ('v)
```

We use it to annotate the structure winter.

```
#let spring = annotate winter;;
spring : tree ('u)
```

We will read a structure with the following function.

```
#let read =
# function
#    'Leaf n → n
#  | 'Node r → r.annot;;
read : tree (pre) → num
```

Of course, it can be applied to the value spring but not to the empty structure winter.

```
#read winter;;
Typechecking error: collision between pre and abs
```

```
#read spring;;
it : num
```

But the function

```
#let rec left =
# function
#   'Leaf n → n
# | 'Node r → left (r.left);;
```

```
left : tree ('u) → num
```

may be applied to both winter and spring.

```
#left winter;;
it : num

#left spring;;
it : num
```

3.4 Extensions

In this section we describe two possible extensions. Both of them have been implemented in a prototype, but not completely formalized yet.

One important motivation for having records was the encoding of some object oriented features into them. But the usual encoding uses recursive types [Car84, Wan89]. An extension of ML with variant types is very easy once we have record types, following the idea of [Rem89], but the extension is actually interesting only if recursive types are allowed.

Thus it would be necessary to extend the results presented here with recursive types. Unification on rational trees in the empty theory is well understood [Hue76, MM82]. In the case of a finite set of labels, the extension of Theorem 2 to rational trees is easy. The infinite case use an equational theory, and unification in its extension with rational trees has no decidable and unitary algorithm in general, even when the original has. But the specificity of the record theory let us conjecture that it can be extended with regular trees.

Another extension which was sketched in [Rem89] partially solves the restrictions due to ML polymorphism. Because subtyping polymorphism goes through lambda abstractions, it could be used to solve some of the examples we failed with. ML type inference with subtyping polymorphism has been first studied by Mitchell in [Mit84] and later by Mishra and Fuh in [FM88, FM89]. The *LET*-case has only been treated in [Jat89]. But as for recursive types, subtyping has never been studied in the presence of an equational theory. Though the general case is certainly difficult, we conjecture that subtyping is compatible with the record theory. We present below an extension with subtyping in the finite case. The extension in the infinite case would be similar but it would depend on the previous conjecture.

It is straight-forward to extend the results of [FM89] to deal with sorted types. It is thus possible to embed the language Π_f into a language with inclusion. In fact, we will start with the language Π_f^* which is the finite case solution but with the signature of the language Π^*. The reason for that could be that this language is more powerful than Π_f, but a more technical reason will appear later. We make very little use of subtyping, for we assume only the atomic coercion

$$pre \subset abs\,,$$

which says that if a field is defined, it can also be considered as undefined. We would assert the following types to the primitives for records:

$$\begin{aligned} null &: \Pi\,(abs.\alpha_1, \ldots abs.\alpha_l) \\ extract^a &: \Pi\,(\varphi_1 \ldots, pre.\alpha \ldots \varphi_l) \rightarrow \alpha \\ new^a &: \Pi\,(\varphi_1, \ldots \varphi_l) \rightarrow \alpha \rightarrow \Pi\,(\varphi_1 \ldots, pre.\alpha, \ldots \varphi_l) \end{aligned}$$

System $\Pi^*_{\subset}$

Note that if the types look the same, they are taken modulo inclusion, and are thus more polymorphic. In this system, we could type

```
let id_eq = field_eq id;;
```

with

```
id_eq : {id : Pre . 'a; 'p} → {id : Pre .'a; 'p} → bool
```

modulo subtyping, which allows the following

```
id_eq car truck;;
```

The field age is implicitly forgotten in truck by the inclusion rules.

But we would still fail with the example

```
choice person car;;
```

for because we could forget the presence of fields but not their types, and there is presently a mismatch between num and string for the old field.

The solution would be to use the system Π_f instead of Π_f^*. But the difficulty is that the inclusion we need is

$$pre(\alpha) \subset abs$$

which is not atomic. Such coercions are not allowed. Type inference with inclusion with non atomic coercions has not been studied yet. The type of primitives for records would be the same as in the system Π_f but modulo this inclusion.

Conclusion

We described a simple, flexible and efficient solution for extending ML with operations on records allowing some kind of inheritance. The solution uses an independent extension of ML with a sorted equational theory over types. An immediate improvement is to allow recursive types needed in many applications of records.

The main limitation of our solution is ML polymorphism, but we conjecture a way of going around. It is not clear yet whether we would want such an extension, for it might not be worth the extra cost in type inference.

This system may be used to add object oriented features, and we hope that ML will regain the attraction that it has been loosing to the benefit of explicitly typed languages.

Acknowledgements

I am grateful for interesting discussions with Peter Buneman, Val Breazu-Tannen and Carl Gunter, and particularly thankful to Benjamin Pierce whose comments on the presentation of this article were very helpful.

References

[Ber88] Bernard Berthomieu. *Une implantation de CCS.* Technical Report 88367, LAAS, 7, Avenue du Colonnel Roche, 31077 Toulouse, France, décembre 1988.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, pages 51–68, Springer Verlag, 1984. Also in Information and Computation, 1988.

[Car86] Luca Cardelli. Amber. In *Combinators and Functional Programming Languages*, pages 21–47, Spinger Verlag, 1986. Proceedings of the 13th Summer School of the LITP.

[Car89] Luca Cardelli. Typefull programming. In *IFIP advanced seminar on Formal Methods in Programming Langage Semantics*, Springer Verlag, 1989.

[CCH*89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-Bounded polymorphism for object oriented programming. In *The Fourth International Conference on Functional Programming Languages and Computer Architecture*, 1989.

[CH89] Guy Cousineau and Gérard Huet. *The CAML Primer.* Institut National de Recherche en Informatique et Automatisme, France, 1989.

[CM89] Luca Cardelli and John C. Mitchell. Operations on records. In *Fifth International Conference on Mathematical Foundations of Programming Semantics*, 1989.

[Cop80] Mario Coppo. An extended polymorphic type system for applicative languages. In *MFCS '80*, pages 194–204, Springer Verlag, 1980.

[Cur87] Pavel Curtis. *Constrained Quantification in Polymorphic Type Analysis* PhD thesis, Cornell, 1987.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17 (4):471–522, 1985.

[FM88] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In *ESOP '88*, pages 94–114, Springer Verlag, 1988.

[FM89] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: closing the theory-practice gap. In *TAPSOFT'89*, 1989.

[HMM90] Robert Harper, David MacQueen, and Robin Milner. *The definition of Standard ML, Version 4.* Laboratory for Foundations of Computer Science, University of Edinburgh, 1990.

[HP90] Robert W. Harper and Benjamin C. Pierce. *Extensible Records Without Subsumption.* Technical Report CMU-CS-90-102, Carnegie Mellon University, Pittsburg, Pensylvania, February 1990.

[Hue76] Gérard Huet. *Résolution d'équations dans les langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris 7, 1976.

[Jat89] Lalita A. Jategaonkar. *ML with Extended Pattern Matching and Subtypes.* Master's thesis, MIT, 545 Technology Square, Cambridge, MA 02139, August 89.

[JM88] Lalita A. Jategaonkar and John C. Mitchell. ML with extended pattern matching and subtypes. In *Proceedings of the 1988 Conference on LISP and Functional Programming*, 1988.

[LC90] Giuseppe Longo and Luca Cardelli. A semantic basis for QUEST. In *Proceedings of the 1990 Conference on LISP and Functional Programming*, 1990.

[Mil80] Robin Milner. A calculus of communicating systems. In *Lecture Notes in Computer Science*, Springer Verlag, 1980.

[Mit84] John C. Mitchell. Coercion and type inference. In *Eleventh Annual Symposium on Principles Of Programming Languages*, 1984.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[OB88] Atsushi Ohori and Peter Buneman. Type inference in a database langage. In *ACM Conference on LISP and Functional Programming*, pages 174–183, 1988.

[OB89] Atsushi Ohori and Peter Buneman. Static type inference for parametric classes. In *Object-Oriented and Programming Systems, Languages, and Applications, Conference Proceedings*, pages 445–456, October 1989.

[Oho90] Atsushi Ohori. *Extending ML Polymorphism to Record Structure.* Technical Report, University of Glasgow, 1990.

[Rem89] Didier Rémy. Records and variants as a natural extension of ML. In *Sixteenth Annual Symposium on Principles Of Programming Languages*, 1989.

[Rem90a] Didier Rémy. *Algèbres Touffues. Application au Typage Polymorphe des Objects Enregistrements dans les Langages Fonctionnels.* PhD thesis, Université de Paris 7, 1990.

[Rem90b] Didier Rémy. *Extending ML Type System with a Sorted Equational Theory.* Technical Report, University of Pensylvania, 1990. To appear. Also in [Rem90a], chapter 3.

[Rem90c] Didier Rémy. *Unification in Syntactic Theories and its Application to the Record Extension of a Free Sorted Algebra.* Technical Report, University of Pensylvania, 1990. To appear. Also in [Rem90a], chapter 2.

[Rey88] John C. Reynolds. *Preliminary Design of the Programming Language Forsythe.* Technical Report CMU-CS-88-159, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1988.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *Second Symposium on Logic In Computer Science*, 1987.

[Wan89] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Fourth Annual Symposium on Logic In Computer Science*, pages 92–97, 1989.

[Wei89] Pierre Weis. *The CAML Reference Manual.* Institut National de Recherche en Informatique et Automatisme, France, 1989.