Type Inference for Records in a Natural Extension of ML

Didier Rémy
INRIA-Rocquencourt*

Abstract

We describe an extension of ML with records where inheritance is given by ML generic polymorphism. All common operations on records but concatenation are supported, in particular the free extension of records. Other operations such as renaming of fields are added. The solution relies on an extension of ML, where the language of types is sorted and considered modulo equations, and on a record extension of types. 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. Some programmers do not like writing the types of their programs by hand. In the ML language for instance, 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 labeled products, a data structure commonly called records, allowing some inheritance between them: records with more labels should be allowed where records with fewer labels are required.

After defining the operations on records and recalling related work, we first review the solution for a finite (and small) set of labels, which was presented in [Rém89], then we extend it to a denumerable set of labels. In the last part we discuss the power and weakness of the solution, we describe some variations, and suggest improvements.

Without records, data structures are built using product types, as in ML, for instance.

```
("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"}}
```

The latter program is definitely more readable than the former. It is also more precise, since components are named. Records can also be used to name several arguments or several results of a function. 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 is the LCS language [Ber88], which is a combination of ML and Milner's CCS [Mil80].

Besides typechecking records, the challenge is to avoid record type declarations and fix size records. Extensible records introduced by Wand [Wan89, CM89] can be built from older records by adding new fields. This feature is the basis of inheritance in the view of objects as records [Wan89, CM89].

*Author's address: INRIA, B.P. 105, F-78153 Le Chesnay Cedex. Email: Didier.Remy@inria.fr

The main operations on records are introduced by examples, using a syntax similar to CAML syntax [CH89, Wei89]. Like variable names, labels do not have particular meanings, though choosing good names (good is subjective) helps 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 following three examples:

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

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

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

Remark that no declaration is required before the use of labels. 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 in person.

All these records have been created in one step. Records can also be build from older ones. For instance, 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 copied 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 newly defined field 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 previously, the operation is not a physical replacement of the vehicle field by a new value. We do not wish the old and the new value of the vehicle field to have the same type. 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 must not be previously defined, and *free* otherwise.

A more general operation than extension is concatenation, which constructs a new record from two previously defined ones, taking the union of their defined fields. If the car has a rusty body but a good engine, one could think of building the hybrid vehicle:

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

This raises the question: what value should be assigned to fields which are defined in both car and truck? When there is a conflict (the same field is defined in both records), priority could be given to the last record. As with free extension, the last record would eventually overwrite fields of the first one. But one might also expect a typechecker to prevent this situation from happening. Although concatenation is less common in the literature, probably because it causes more trouble, it seems interesting in some cases. Concatenation is used in the standard ML language [HMT91] when a structure is opened and extended with another one. In the LCS language, 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 concatenation.

The constructions described above are not exhaustive but are the most common ones. We should also mention the permutation, renaming and erasure of fields. We described how to build records, but of course we also want to read them. There is actually a unique construction for this purpose.

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

Accessing some field a of a record x can be abstracted over x, but not over a: Labels are not values and there is no function which could take a label as argument and would access the field of some

fixed 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 possessing the field to access. 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 eq below should of course accept arguments of different types provided they have an id field of the same type.

```
let eq x y = equal x.id y.id;;                      eq car truck;;
```

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

Related work

Luca Cardelli has always claimed that functional languages should have record operations. In 1986, when he designed Amber, his choice was to provide the language with records rather than polymorphism. Later, he introduced bounded quantification in the language FUN , which he extended to higher order bounded quantification in the language QUEST. Bounded quantification is an extension of ordinary quantification where quantified variables range in the subset of types that are all subtypes of the bound. The subtyping relation is a lattice on types. In this language, subtyping is essential for having some inheritance between records. 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 trend seems to be the simplification rather than the enrichment of existing systems [LC90, HP90, Car91]. For instance, an interesting goal was to remove the subtype relation in bounded quantification [HP90]. Records have also been formulated with explicit labeled conjunctive types in the language Forsythe [Rey88].

In contrast, records in implicitly typed languages have been less studied, and the proposed extensions of ML are still very restrictive. The language Amber [Car84, Car86] is monomorphic and inheritance is obtained by type inclusion. A major step toward combining records and type inference has been Wand's proposal [Wan87] where inheritance is obtained from ML generic polymorphism. Though type inference is incomplete for this system, it remains a reference, for it was the first concrete proposal for extending ML with records having inheritance. The year after, complete type inference algorithms were found for a strong restriction of this system [JM88, OB88]. The restriction only allows the strict extension of a record. Then, the author proposed a complete type inference algorithm for Wand's system [Rém89], but it was formalized only in the case of a finite set of labels (a previous solution given by Wand in 1988 did not admit principal types but complete sets of principal types, and was exponential in size 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 has been addressed in [Rém90], which we review in this article.

1 A simple solution when the set of labels is finite

Though the solution below 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 decent solution in cases where only few labels are needed. And it will emphasize a

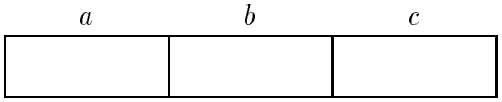
[1] It can be understood as an "append" on association lists in lisp compared to the "with" operation which should be understood as a "cons".

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).

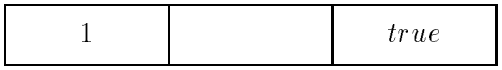
We will sketch the path from Wand's proposal to this solution, for it may be of some interest to describe the method which we think could be applied in other situations. As intuitions are rather subjective, and ours may not be yours, the section 1.1 can be skipped whenever it does not help.

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:



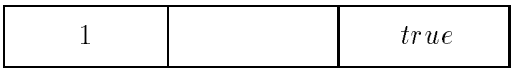
Defining a record is the same as filling some of the fields with values. For example, we will put the values 1 and *true* in the a and c fields respectively and leave the b field undefined.



Typechecking means forgetting some information about values. For instance, it does not distinguish two numbers but 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 record values with partial functions from labels ($\mathcal{L}$) to types ($\mathcal{T}$), that is, elements of $\mathcal{L} \longrightarrow\!\!\!\!\!\!\circ\ \mathcal{T}$. We first make record types total functions on labels using an explicitly undefined constant *abs* ("absent"): $\mathcal{L} \longrightarrow \mathcal{T} \cup \{abs\}$. In fact, we replace the union by the sum $pre(\mathcal{T}) + abs$. Finally, we decompose record types as follows:

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

The first function is an ordering from $\mathcal{L}$ to the segment $[1, Card(\mathcal{L})]$ 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 $Card(\mathcal{L})$ of types in $pre(\mathcal{T}) + abs$. The previous example is typed by



$$\Pi(\ pre(num)\ ,\quad abs\quad ,\ pre(bool)\ )$$

A function $\_.a$ reading the a field accepts as argument any record having the a field defined with a value M, and returns M. The a field of the type of the argument must be $pre(\tau)$ if τ is the type of M. We do not care whether other fields are defined or not, so their types may be anything. We choose to represent them by variables θ and ε. The result has type α.

$$\_.a : \Pi(pre(\alpha), \theta, \varepsilon) \rightarrow \alpha$$

1.2 A formulation

We are given a collection of symbols $\mathcal{C}$ with their arities $(\mathcal{C}^n)_{n \in N}$ that contains at least an arrow symbol $\rightarrow$ of arity 2, a unary symbol *pre* and a nullary symbol *abs*. We are also given two sorts *type* and *field*. The signature of a symbol is a sequence of sorts, written ι for a nullary symbol and

$\iota_1 \ldots \otimes \iota_n \Rightarrow \iota$ for a symbol of arity n. The signature $\mathcal{S}$ is defined by the following assertions (we write $\mathcal{S} \vdash f :: \iota$ for $(f, \iota) \in \mathcal{S}$):

$$\begin{array}{l} \mathcal{S} \vdash \mathit{pre} :: \mathit{type} \Rightarrow \mathit{field} \\ \mathcal{S} \vdash \mathit{abs} :: \mathit{field} \\ \mathcal{S} \vdash \Pi :: \mathit{field}^{\mathit{card}(\mathcal{L})} \Rightarrow \mathit{type} \\ \mathcal{S} \vdash f :: \mathit{type}^n \Rightarrow \mathit{type} \qquad\qquad f \in \mathcal{C}^n \setminus \{\mathit{pre}, \mathit{abs}, \Pi\} \end{array}$$

The language of types is the free sorted algebra $\mathcal{T}(\mathcal{S}, \mathcal{V})$. 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 these rules is still decidable and admits principal typings (see next section for a precise formulation). In this language, we assume the following primitive environment:

$$\begin{array}{c} \{\} : \Pi\,(\mathit{abs}, \ldots \mathit{abs}) \\ \_.a : \Pi\,(\theta_1 \ldots, \mathit{pre}\,(\alpha), \ldots \theta_l) \to \alpha \\ \{\_ \mathit{\ with\ } a = \_\} : \Pi\,(\theta_1, \ldots \theta_l) \to \alpha \to \Pi\,(\theta_1 \ldots, \mathit{pre}\,(\alpha), \ldots \theta_l) \end{array}$$

Basic constants for $\Pi\mathrm{ML}_{fin}$

The constant $\{\}$ is the empty record. The $\_.a$ constant reads the a field from its argument, we write $r.a$ the application $(\_.a)\ r$. Similarly $\{r \mathit{\ with\ } a = M\}$ extends the records r on label a with value M.

2 Extension to large records

Though the previous solution is 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 the size of record types is proportional to the size of this set — even for the type of the null record, which has no field defined. When a local use of records is needed, labels may be fewer than ten and the solution works perfectly. But in large systems 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 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. countable, set of labels. From a theoretical point of view, it is the only way to avoid reasoning outside of the formalism and 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. A better 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 discussion above and describe the intuitions. Then we 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 with sorts and equations on types. 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 $\{a = 1\ ;\ b = \mathit{true}\}$ and f the function that reads the a field. Assuming f has type $\tau \to \tau'$ and r has type σ, f can be applied to r if the two types τ and σ are unifiable. In our example, we have

$$\begin{array}{l} \tau = \Pi\,(a : \mathit{pre}\,(\alpha)\ ;\ b : \theta_b)\,, \\ \sigma = \Pi\,(a : \mathit{pre}\,(\mathit{num})\ ;\ b : \mathit{pre}\,(\mathit{bool}))\,, \end{array}$$

and τ' is equal to α. The unification of τ and σ is done field by field and their most general unifier is:

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

If we had one more label c, the types τ and σ would be

$$\begin{aligned} \tau &= \Pi\,(a : pre\,(\alpha)\ ;\ b : \theta_b\ ;\ c : \theta_c)\,, \\ \sigma &= \Pi\,(a : pre\,(num)\ ;\ b : pre\,(bool)\ ;\ c : abs\,)\,. \end{aligned}$$

and their most general unifier

$$\begin{cases} \alpha \mapsto num \\ \theta_b \mapsto pre\,(bool) \\ \theta_c \mapsto abs \end{cases}$$

We can play again with one more label d. The types would be

$$\begin{aligned} \tau &= \Pi\,(a : pre\,(\alpha)\ ;\ b : \theta_b\ ;\ c : \theta_c\ ;\ d : \theta_d)\,, \\ \sigma &= \Pi\,(a : pre\,(num)\ ;\ b : pre\,(bool)\ ;\ c : abs\ \ ;\ d : abs\,)\,. \end{aligned}$$

whose most general unifier is:

$$\begin{cases} \alpha \mapsto num \\ \theta_b \mapsto pre\,(bool) \\ \theta_c \mapsto abs \\ \theta_d \mapsto abs \end{cases}$$

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

$$\begin{aligned} \tau &= \Pi\,(a : pre\,(\alpha)\ ;\ b : \theta_b\ ;\ \infty : \theta_\infty)\,, \\ \sigma &= \Pi\,(a : pre\,(num)\ ;\ b : pre\,(bool)\ ;\ \infty : abs\,)\,. \end{aligned}$$

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

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

The most general unifier can be computed without developing this expression, thus allowing the set of labels to be infinite. We summarize the successive steps studied above in this figure:

Labels	a	b	c	d	∞
τ	$pre\,(\alpha)$	θ_b	θ_c	θ_d	θ_∞
σ	$pre\,(num)$	$pre\,(bool)$	abs	abs	abs
$\tau \wedge \sigma$	$pre\,(num)$	$pre\,(bool)$	abs	abs	abs

This approach is so intuitive that it seems very simple. There is a difficulty though, due to the sharing between 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 τ into a copy τ' (the type of the extracted field) and another copy τ'' (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 term $f(\alpha)$, assuming that we have already extracted field ℓ from α, i.e. we have $f(\ell : \beta \ ; \ \gamma)$, we now want to replace it by $\ell : f(\alpha) \ ; \ f(\gamma)$. The solution is simply to ask it to be true, that is, to assume the axiom

$$f(\ell : \beta \ ; \ \gamma) = \ell : f(\alpha) \ ; \ f(\gamma)$$

for every given symbol f but Π.

2.2 Extending a free algebra with a record algebra

The intuitions of previous sections are formalized by the algebra of record terms. The algebra of record terms is introduced for an arbitrary free algebra; record types are an instance. The record algebra was introduced in [Rém90] and revisited in [Rém92b]. We summarize it below but we recommend [Rém92b] for a more thorough presentation.

We are given a set of variables $\mathcal{V}$ and a set of symbols $\mathcal{C}$ with their arities $(\mathcal{C}_n)_{n \in N}$.

Raw terms

We call *unsorted record terms* the terms of the free unsorted algebra $\mathcal{T}'(\mathcal{D}', \mathcal{V})$ where $\mathcal{D}'$ is the set of symbols composed of $\mathcal{C}$ plus a unary symbol Π and a collection of projection symbols $\{(\ell : \_ \ ; \ \_) \mid \ell \in \mathcal{L}\}$ of arity two. Projection symbols associate to the right, that is $(a : \tau \ ; \ b : \sigma \ ; \ \tau')$ stands for $(a : \tau \ ; \ (b : \sigma \ ; \ \tau'))$.

Example 1 The expressions

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

are raw terms. In section 2.4 we will consider the former as a possible type for the record $\{a = 1 \ ; \ c = true\}$ but we will not give a meaning to the latter. There are too many raw terms. The raw term $\{a : \alpha \ ; \ \chi\} \rightarrow \chi$ must also be rejected since the template composed of the raw variable χ should define the a field on the right but should not on the left. We define record terms using sorts to constrain their formation. Only a few of the raw terms will have associated record terms.

Record terms

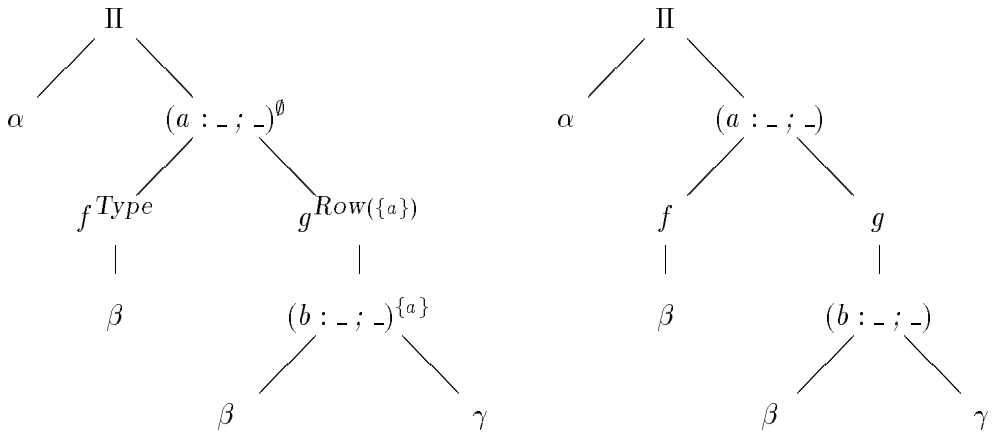
Let $\mathcal{L}$ be a denumerable set of labels. Let $\mathcal{K}$ be composed of a sort *type* , and a finite collection of sorts $(row\,(L))$ where L range over finite subsets of labels. Let $\mathcal{S}$ be the signature composed of the following symbols given with their sorts:

$$\begin{array}{ll}
\mathcal{S} \vdash \Pi :: Row(\emptyset) \Rightarrow Type & \\
\mathcal{S} \vdash f^K :: K^n \Rightarrow K & f \in \mathcal{C}^n, K \in \mathcal{K} \\
\mathcal{S} \vdash (\ell^L : _ \ ; \ _) :: Type \otimes Row(L \cup \{\ell\}) \Rightarrow Row(L) & \ell \in \mathcal{L}, L \in \mathcal{P}_{fin}(\mathcal{L} \setminus \{\ell\})
\end{array}$$

The superscripts are parts of symbols, so that the signature $\mathcal{S}$ is not overloaded, that is, every symbol has a unique signature. We write $\mathcal{D}$ the set of symbols in $\mathcal{S}$.

Definition 1 *Record terms* are the terms of the free sorted algebra $\mathcal{T}(\mathcal{S}, \mathcal{V})$. □

Example 2 The left term below is a record term. On the right, we drew a raw term with the same structure.



Script erasure

To any record term, we associate the raw term obtained by erasing all superscripts of symbols. Conversely, for any raw term τ', and any sort ι there is at most one record term whose erasure is τ'. Thus any record term τ of sort ι is completely defined by its erasure τ' and the sort ι. In the rest of the paper we will mostly use this convention. Moreover we usually drop the sort whenever it is implicit from context.

Example 3 The erasure of

$$\Pi\left(a^{\emptyset} : f^{Type}(g^{Type}) \; ; \left(c^{\{a\}} : f^{Type}(\alpha) \; ; \, h^{Row(\{a,c\})}\right)\right)$$

is the raw term

$$\Pi\,(a : f(g) \; ; \; c : f(\alpha) \; ; \; h)$$

There is no record term whose erasure would be

$$\Pi\,(a : f(b : g \; ; \; \alpha) \; ; \; h)$$

Record algebra

The permutation and the extraction of fields in record terms will be obtained by equations, of left commutativity and distributivity respectively. Precisely, let E be the set of axioms

- Left commutativity. For any labels a and b and any finite subset of labels L that do not contain a and b,
$$a^L : \alpha \; ; \left(b^{L\cup\{a\}} : \beta \; ; \; \gamma\right) = b^L : \beta \; ; \left(a^{L\cup\{b\}} : \alpha \; ; \; \gamma\right)$$
- Distributivity. For any symbol f, any label a and any finite subset of labels L that do not contain a,
$$f^{Row(L)}(a^L : \alpha_1 \; ; \; \beta_1, \ldots a^L : \alpha_p \; ; \; \beta_p) = a^L : f^{Type}(\alpha_1, \ldots \alpha_p) \; ; \; f^{Row(L\cup\{a\})}(\beta_1, \ldots \beta_p)$$

With the raw notation the equations are written:

- Left commutativity. At any sort $row(L)$, where L does not contain labels a and b:

$$a : \alpha \ ; \ (b : \beta \ ; \ \gamma) = b : \beta \ ; \ (a : \alpha \ ; \ \gamma)$$

- Distributivity. At any sort $row(L)$ where L does not contain label a, and for any symbol f:

$$f(a : \alpha_1 \ ; \ \beta_1, \ldots a : \alpha_p \ ; \ \beta_p) = a : f(\alpha_1, \ldots \alpha_p) \ ; \ f(\beta_1, \ldots \beta_p)$$

All axioms are regular, that is, the set of variables of both sides of equations are always identical.

Example 4 In the term

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

we can replace abs by $b : abs \ ; abs$ using distributivity, and use left commutativity to end with the term:

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

In the term

$$\Pi\,(a : pre\,(\alpha) \ ; \ \theta)$$

we can substitute θ by $b : \theta_b \ ; \ c \ ; \ \theta_c \ ; \ \varepsilon$ to get

$$\Pi\,(a : pre\,(\alpha) \ ; \ b : \theta_b \ ; \ c : \theta_c \ ; \ \varepsilon)$$

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

Definition 2 The algebra of record terms is the algebra $\mathcal{T}(\mathcal{S}, \mathcal{V})$ modulo the equational theory E, written $\mathcal{T}(\mathcal{S}, \mathcal{V})/E$. □

Unification in the algebra of record terms has been studied in [Rém92b].

Theorem 1 *Unification in the record algebra is decidable and unitary (every solvable unification problem has a principal unifier).*

A unification algorithm is given in the appendix.

Instances of record terms

The construction of the record algebra is parameterized by the initial set of symbols $\mathcal{C}$, from which the signature $\mathcal{S}$ is deduced. The signature $\mathcal{S}$ may also be restricted by a signature $\mathcal{S}'$ that is compatible with the equations E, that is, a signature $\mathcal{S}'$ such that for all axioms r and all sorts ι of $\mathcal{S}'$,

$$\mathcal{S}' \vdash r^l :: \iota \iff \mathcal{S}' \vdash r^r :: \iota$$

The algebra $(\mathcal{T}/E)\restriction\mathcal{S}'$ and $(\mathcal{T}\restriction\mathcal{S}')/(E\restriction\mathcal{S}')$ are then isomorphic, and consequently unification in $(\mathcal{T}\restriction\mathcal{S}')/(E\restriction\mathcal{S}')$ is decidable and unitary, and solved by the same algorithm as in $\mathcal{T}/E$. The $\mathcal{S}'$-record algebra is the restriction $\mathcal{T}(\mathcal{S}, \mathcal{V})\restriction\mathcal{S}'$ of the record algebra by a compatible signature $\mathcal{S}'$.

We now consider a particular instance of record algebra, where fields are distinguished from arbitrary types, and structured as in section 1. The signature $\mathcal{S}'$ distinguishes a constant symbol abs and a unary symbol pre in $\mathcal{C}$, and is defined with two sorts $type$ and $field$:

$$\begin{array}{ll}
\mathcal{S}' \vdash \Pi :: field \ \Rightarrow \ type & \\
\mathcal{S}' \vdash abs^{\,\iota} :: field & \iota \in \mathcal{K} \\
\mathcal{S}' \vdash pre \ :: type \ \Rightarrow field & \\
\mathcal{S}' \vdash f^{Type} :: type^{\,n} \Rightarrow type & f \in \mathcal{C}^n \setminus \{abs\,, pre\,, \Pi\} \\
\mathcal{S}' \vdash (\ell^L : _ \ ; \ _) :: field \otimes field \ \Rightarrow field & \ell \in \mathcal{L}, L \in \mathcal{P}_{fin}(\mathcal{L} \setminus \{\ell\})
\end{array}$$

The signature $\mathcal{S}'$ is compatible with the equations of the record algebra. We call *record types* the $\mathcal{S}'$-record algebra.

In fact, record types have a very simple structure. Terms of the sort $Row(L)$ are either of depth 0 (reduced to a variable or a symbol) or are of the form $(a : \tau \ ; \tau')$. By induction, they are always of the form

$$(a_1 : \tau_1 \ ; \ldots a_p : \tau_p \ ; \sigma)$$

where σ is either abs or a variable, including the case where p is zero and the term is reduced to σ. Record types are also generated by the pseudo-BNF grammar:

$$\begin{array}{lll} \tau ::= \alpha \mid \tau \to \tau \mid \Pi\rho^{\emptyset} & & \text{types} \\ \rho^L ::= \chi^L \mid abs^{\,L} \mid a : \varphi \ ; \rho^{L\cup\{a\}} & a \notin L & \text{rows} \\ \varphi ::= \theta \mid abs \mid pre\,(\tau) & & \text{fields} \end{array}$$

where α, β, γ and δ are type variables, χ, π and ξ are row variables and θ and ε are field variables. We prefer the algebraic approach which is more general.

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. A regular theory is one whose left and right hand sides of axioms always have the same set of variables. For any term τ of $\mathcal{T}/E$ we write $\mathcal{V}(\tau)$ for the set of its variables. We privilege a sort $Type$.

The addition of a sorted equational theory to the types of ML has been studied in [Rém90, Rém92a]. We recall here the main definitions and results. The language ML that we study is lambda-calculus extended with constants and a LET construct in order to mark some of the redexes, namely:

$M ::=$	Terms	M, N
x	Variable	x, y
$\mid c$	Constant	c
$\mid \lambda\, x\,.\, M$	Abstraction	
$\mid M\ M$	Application	
$\mid$ *let* $x = M$ *in* M	Let binding	

The letter W ranges over finite set of variables. Type schemes are pairs noted $\forall\, W \cdot \tau$ of a set of variables and a term τ. The symbol $\forall$ is treated as a binder and we consider type schemes equal modulo α-conversion. The sort of a type scheme $\forall\, W \cdot \tau$ is the sort of τ. Contexts as sequences of assertions, that is, pairs of a term variable and a type. We write $\mathcal{A}$ the set of contexts.

Every constant c comes with a closed type scheme $\forall\, W \cdot \tau$, written $c : \forall\, W \cdot \tau$. We write B the collection of all such constant assertions. We define a relation $\vdash$ on $\mathcal{A} \times \mathrm{ML} \times \mathcal{T}$ and parameterized by B as the smallest relation that satisfies the following rules:

$$\frac{x : \forall\, W \cdot \tau \in A \qquad \mu : W \to \mathcal{T}}{A \vdash_S x : \mu(\tau)} \ (\textsc{Var-Inst}) \qquad \frac{c : \forall\, W \cdot \tau \in B \qquad \mu : W \to \mathcal{T}}{A \vdash_S c : \mu(\tau)} \ (\textsc{Const-Inst})$$

$$\frac{A[x : \tau] \vdash M : \sigma \qquad \tau \in \mathcal{T}}{A \vdash \lambda\, x\,.\, M : \tau \to \sigma} \ (\textsc{Fun}) \qquad \frac{A \vdash M : \sigma \to \tau \qquad A \vdash N : \sigma}{A \vdash M\ N : \tau} \ (\textsc{App})$$

$$\frac{A \vdash_S M : \tau \qquad A[x : \forall\, W \cdot \tau] \vdash_S N : \sigma \qquad W \cap \mathcal{V}(A) = \emptyset}{A \vdash_S \mathit{let}\ x = M\ \mathit{in}\ N : \sigma} \ (\textsc{Let-Gen})$$

$$\frac{A \vdash M : \sigma \qquad \sigma =_E \tau}{A \vdash M : \tau} \ (\textsc{Equal})$$

They are the usual rules for ML except the rule EQUAL that is added since the equality on types is taken modulo the equations E.

A typing problem is a triple of $\mathcal{A} \times \mathrm{ML} \times \mathcal{T}$ written $A \triangleright M : \tau$. The application of a substitution μ to a typing problem $A \triangleright M : \tau$ is the typing problem $\mu(A) \triangleright M : \mu(\tau)$, where substitution of a context is understood pointwise and only affects the type part of assertions. A solution of a typing problem $A \triangleright M : \tau$ is a substitution μ such that $\mu(A) \vdash M : \mu(\tau)$. It is principal if all other solutions are obtained by left composition with μ of an arbitrary solution.

Theorem 2 (principal typings) *If the sorted theory $\mathcal{T}/E$ is regular and its unification is decidable and unitary, then the relation $\vdash$ admits principal typings, that is, any solvable typing problem has a principal solution.*

Moreover, there is an algorithm that given a typing problem computes a principal solution if one exists, or returns failure otherwise.

An algorithm can be obtained by replacing free unification by unification in the algebra of record terms in the core-ML type inference algorithm. A clever algorithm for type inference is described in [Rém92b].

2.4 Typechecking record operations

Using the two preceding results, we extend the types of ML with record types assuming given the following basic constants:

$$\begin{gathered} \{\} : \Pi\,(abs\,) \\ \_.a : \Pi\,(a : pre\,(\alpha)\ ;\ \theta) \to \alpha \\ \{\_\ with\ a = \_\} : \Pi\,(a : \theta\ ;\ \chi) \to \alpha \to \Pi\,(a : pre\,(\alpha)\ ;\ \chi) \end{gathered}$$

Basic constants for ΠML

There are countably many constants. We write $\{a_1 = x_1\ ;\ \ldots a_n = x_n\}$ as syntactic sugar for:

$$\{\{a_1 = x_1\ ;\ \ldots a_{n-1} = x_{n-1}\}\ with\ a_n : x_n\}$$

We illustrate this system by examples in the next section.

The equational theory of record types is regular, and has a decidable and unitary unification. It follows from theorems 2 and 1 that the typing relation of this language admits principal typings, and has a decidable type inference algorithm.

3 Programming with records

We first show on simple examples how most of the constructions described in the introduction are typed, then we meet the limitations of this system. Some of them can be cured by slightly improving the encoding. Finally, we propose and discuss some further extensions.

3.1 Typing examples

A typechecking prototype has been implemented in the CAML language. It was used to automatically type all the examples presented here and preceded by the # character. In programs, type variables are printed according to their sort in $\mathcal{S}'$. Letters $\boldsymbol{\chi}$, $\boldsymbol{\pi}$ and $\boldsymbol{\xi}$ are used for field variables and letters $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, etc. are used for variables of the sort *type* . We start with simple examples and end with a short program.

Simple record values can be built as follows:

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

#let truck = {name = "Blazer"; id = 6587867567};;
truck:Π (name:pre (string); id:pre (num); abs)

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

Each field defined with a value of type τ is significant and typed with $pre\,(\tau)$. Other fields are insignificant, and their types are gathered in the template abs. The record person can be extended with a new field vehicle:

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

This is possible whether this field was previously undefined as above, or defined as in:

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

The concatenation of two records is not provided by this system.

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

```
#let age x = x.age;;
age:Π (age:pre (α); χ) → α

#let id x = x.id;;
id:Π (id:pre (α); χ) → α
```

The accessed field must be defined with a value of type α, so it has type pre (α), and other fields may or may not be defined; they are described by a template variable χ. The returned value has type α. As any value, age can be sent as an argument to another function:

```
#let car_info field = field car;;
car_info:(Π (name:pre (string); id:pre (num); age:pre (string); abs) → α) → α

#car_info age;;
it:string
```

The function equal below takes two records both possessing an id field of the same type, and possibly other fields. For simplicity of examples we assume given a polymorphic equality equal.

```
#let eq x y = equal x.id y.id;;
eq:Π (id:pre (α); χ) → Π (id:pre (α); π) → bool

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

We will show more examples in section 3.3.

3.2 Limitations

There are two sorts of limitations, one is due to the encoding method, the other one results from ML generic polymorphism. The only source of polymorphism in record operations is generic polymorphism. A field defined with a value of type τ in a record object is typed by $pre\,(\tau)$. Thus, once a field has been 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:α → α → α
```

Typechecking fails with:

```
#choice car truck;;
Typechecking error:collision between pre (string) and abs
```

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

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

which should be equivalent to the program

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

may actually be useful. We will partially solve this problem in section 3.3. A natural generalization of the eq function defined above is to abstract over the field that is used for testing equality

```
#let field_eq field x y = equal (field x) (field y);;
field_eq:(α → β) → α → α → bool
```

It is enough general to test equality on other values than records. We get a function equivalent to the program eq defined in section 3.1 by applying field_eq to the function id.

```
#let id_eq = field_eq id;;
id_eq:Π (id:pre (α); χ) → Π (id:pre (α); χ) → bool

#id_eq car truck;;
Typechecking error:collision between pre (string) and abs
```

The last example fails. This is not surprising since field is bound by a lambda in field_eq, and therefore its two instances have the same type, and so have both arguments x and y. In eq, the arguments x and y are independent since they are two instances of id. This is nothing else but ML generic polymorphism restriction. We emphasize that, as record polymorphism is entirely based on generic polymorphism, the restriction applies drastically to records.

3.3 Flexibility and Improvements

The method for typechecking records is very flexible: the operations on records have not been fixed at the beginning, but at the very end. They are parameters that can vary in many ways.

The easiest modification is changing the types of basic constants. For instance, asserting that $\{\_ \; with \; a = \_\}$ comes with type scheme:

$$\{\_ \; with \; a = \_\} : \Pi\,(a : abs \; ; \chi) \to \alpha \to \Pi\,(a : pre\,(\alpha) \; ; \chi)$$

makes the extension of a record with a new field possible only if the field was previously undefined. This slight change gives exactly the strict version 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 strict construction may be easier to implement and more efficient.

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

$$\_ \setminus a : \Pi\,(a : \theta \; ; \chi) \to \Pi\,(a : abs \; ; \chi)\,,$$

In fact, the constant $\{\_ \; with \; a = \_\}$ is not primitive. It should be replaced by the strict version:

$$\{\_ \; with \; !a = \_\} : \Pi\,(a : abs \;;\; \chi) \rightarrow \alpha \rightarrow \Pi\,(a : pre\,(\alpha) \;;\; \chi)\,,$$

and the $\_\backslash a$ constant, since the original version is the composition $\{\_ \setminus a \; with \; !a = \_\}$. Our encoding also allows typing a function that renames fields

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

The renamed field may be undefined. In the result, it is no longer accessible. A more primitive function would just exchanges two fields

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

whether they are defined or not. Then the *rename* constant is simply the composition:

$$(\_ \setminus a) \circ exchange^{a \leftrightarrow b}$$

More generally, the decidability of type inference does not depend on the specific signature of the *pre* and *abs* type symbols. The encoding of records can be revised. We are going to illustrate this by presenting another variant for type-checking records.

We suggested that a good type system should allow some polymorphism on records values themselves. We recall the example that 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. We would like the result to have a type with *abs* on this field to guarantee that it will not be accessed, but common, compatible fields should remain accessible. The idea is that a defined field should be seen as undefined whenever needed. From the point of view of types, this would require that a defined field with a value of type τ should be typed with both $pre\,(\tau)$ and abs.

Conjunctive types [Cop80] could possibly solve this problem, but they are undecidable in general. Another attempt is to make *abs* of arity 1 by replacing each use of *abs* by $abs\,(\alpha)$ where α is a generic variable. However, it is not possible to write $\forall\,\theta \cdot \theta(\tau)$ where θ ranges over *abs* and *pre*. The only possible solution is to make *abs* and *pre* constant symbols by introducing an infix field symbol "." and write $abs\,.\alpha$ and $pre\,.\alpha$ instead of $abs\,(\alpha)$ and $pre\,(\alpha)$. It is now possible to write $\forall\,\varepsilon \cdot (\varepsilon.\tau)$. Formally, the signature $\mathcal{S}'$ is replaced by the signature $\mathcal{S}''$ given below, with a new sort *flag*:

$$\begin{array}{ll}
\mathcal{S}'' \vdash \Pi :: field \Rightarrow type & \\
\mathcal{S}'' \vdash abs^{\,\iota} :: flag & \iota \in \mathcal{K} \\
\mathcal{S}'' \vdash pre^{\,\iota} :: flag & \iota \in \mathcal{K} \\
\mathcal{S}'' \vdash .^{\iota} :: flag \otimes type \Rightarrow field & \iota \in \mathcal{K} \\
\mathcal{S}'' \vdash f^{Type} :: type^{\,\varrho(f)} \Rightarrow type & f \in \mathcal{C} \setminus \{abs\,, pre\,, .\} \\
\mathcal{S}'' \vdash (\ell^{L} : _\;;\; _) :: field \otimes field \Rightarrow field & \ell \in \mathcal{L}, L \in \mathcal{P}_{fin}(\mathcal{L} \setminus \{\ell\})
\end{array}$$

Record constants now come with the following type schemes:

$$\begin{array}{c}
\{\} : \Pi\,(abs\,.\alpha) \\
_.a : \Pi\,(a : pre\,.\alpha \;;\; \chi) \rightarrow \alpha \\
\{_ \; with \; a = _\} : \Pi\,(a : \theta \;;\; \chi) \rightarrow \alpha \rightarrow \Pi\,(a : \varepsilon.\alpha \;;\; \chi)
\end{array}$$

Basic constants for $\Pi\mathrm{ML}'$

It is easy to see that system $\Pi\mathrm{ML}'$ is more general than system $\Pi\mathrm{ML}$; any expression typeable in the system $\Pi\mathrm{ML}$ is also typeable in the system $\Pi\mathrm{ML}'$: replacing in a proof all occurrences of *abs*

by $abs.\alpha$ and all occurrence of $pre\,(\tau)$ by $pre.\tau$ (where α does not appear in the proof), we obtain a correct proof in IIML′.

We show the types in the system IIML′ of some of previous examples. Flag variables are written ε, ζ and η. Building a record creates a polymorphic object, since all fields have a distinct *flag* variable:

```
#let car = {name = "Toyota"; age = "old"; id = 7866};;
car:Π (name:ε.string; id:ζ.num; age:η.string; abs.α)

#let truck = {name = "Blazer"; id = 6587867567};;
truck:Π (name:ε.string; id:ζ.num; abs.α)
```

Now these two records can be merged,

```
#choice car truck;;
it:Π (name:ε.string; id:ζ.num; age:abs.string; abs.α)
```

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 that have stayed in fields. Thus, the type system IIML′ rejects the program:

```
#let person = {name = "Tim"; age = 31; id = 5656787};;
person:Π (name:ε.string; id:ζ.num; age:η.num; abs.α)

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

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

$$\_ \setminus a : \Pi\,(a : \theta\ ;\ \chi) \rightarrow \Pi\,(a : abs.\alpha\ ;\ \chi)\,,$$

then we can turn around the failure above by explicitly forgetting label age in at least one record

```
#choice (car \ age) person;;
it:Π (age:abs.num; name:ε.string; id:ζ.num; abs.α)

#choice car (person \ age);;
it:Π (age:abs.string; name:ε.string; id:ζ.num; abs.α)

#choice (car \ age) (person \ age);;
it:Π (age:abs.α; name:ε.string; id:ζ.num; abs.β)
```

A more realistic example illustrates the ability to add annotations on data structures and type the presence of these annotations. The example is run into the system IIML′, where we assume given an infix addition + typed with num → num → num.

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

The variable ε 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 (ε) → tree (ζ) * num
```

```
#let annotate x = match annotation x with y,_ → y;;
annotate:tree (ε) → tree (ζ)
```

We use it to annotate the structure winter.

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

We will read a structure with the following function.

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

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 following function may be applied to both winter and spring:

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

```
#left winter;;
it:num
```

```
#left spring;;
it:num
```

3.4 Extensions

In this section we describe two possible extensions. The two 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 easy once we have record types, following the idea of [Rém89], but the extension is interesting essentially if recursive types are allowed.

Thus it would be necessary to extend the results presented here with recursive types. Unification on rational trees without equations 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 uses an equational theory, and unification in the extension of first order equational theory to rational trees has no decidable and unitary algorithm in general, even when the original theory has one. But the simplicity of the record theory lets us conjecture that it can be extended with regular trees.

Another extension, which was sketched in [Rém89], partially solves the restrictions due to ML polymorphism. Because subtyping polymorphism goes through lambda abstractions, it could be used to type some of the examples that were wrongly rejected. ML type inference with subtyping

polymorphism has been first studied by Mitchell in [Mit84] and later by Mishra and Fuh [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. Although the general case of merging subtyping with an equational theory is certainly difficult, we believe that subtyping is compatible with the axioms of the algebra of record types. We discuss below the extension with subtyping in the finite case only. The extension in the infinite case would be similar, but it would rely on the previous conjecture.

It is straightforward to extend the results of [FM89] to deal with sorted types. It is thus possible to embed the language $\Pi\mathrm{ML}_{fin}$ into a language with subtypes $\Pi\mathrm{ML}_{\subset}$. In fact, we use the language $\Pi\mathrm{ML}'_{\subset}$ that has the signature of the language $\Pi\mathrm{ML}'$ for a technical reason that will appear later. The subtype relation we need is closed structural subtyping. Closed[2] structural subtyping is defined relatively to a set of atomic coercions as the smallest E-reflexive (i.e. that contains $=_E$) and transitive relation $\subset$ that contains the atomic coercions and that satisfies the following rules [FM89]:

$$\frac{\sigma \subset \tau \qquad \tau' \subset \sigma'}{\tau \to \tau' \subset \sigma \to \sigma'}$$

$$\frac{\tau_1 \subset \sigma_1, \ldots \tau_p \subset \sigma_p}{f(\tau_1, \ldots \tau_p) \subset f(\sigma_1, \ldots \sigma_p)} \qquad f \in \mathcal{C} \setminus \{\to\}$$

In $\Pi\mathrm{ML}'_{\subset}$, we consider the unique atomic coercion $pre \subset abs$. It says that if a field is defined, it can also be view as undefined. We assign the following types to constants:

$$\begin{gathered}
\{\} : \Pi\,(abs\,.\alpha_1, \ldots abs\,.\alpha_l) \\
_.a : \Pi\,(\theta_1 \ldots, pre\,.\alpha \ldots \theta_l) \to \alpha \\
\{_ \; with \; a = _\} : \Pi\,(\theta_1, \ldots \theta_l) \to \alpha \to \Pi\,(\theta_1 \ldots, pre\,.\alpha, \ldots \theta_l)
\end{gathered}$$

Basic constants for $\Pi\mathrm{ML}'_{\subset}$

If the types look the same as without subtyping, they are taken modulo subtyping, and are thus more polymorphic. In this system, the program

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

is typed with:

id_eq : {id : pre.α; χ} → {id : pre.α; χ} → bool

This allows the application modulo subtyping `id_eq car truck`. The field `age` is implicitly forgotten in `truck` by the inclusion rules. However, we still fail with the example `choice person car`. The presence of fields can be forgotten, yet their types cannot, and there is a mismatch between `num` and `string` in the `old` field of both arguments. A solution to this failure is to use the signature $\mathcal{S}'$ instead of $\mathcal{S}''$. However the inclusion relation now contains the assertion $pre\,(\alpha) \subset abs$ which is not atomic. Such coercions do not define a structural subtyping relation. Type inference with non structural inclusion has not been studied successfully yet and it is surely difficult (the difficulty is emphasized in [Rém89]). The type of primitives for records would be the same as in the system $\Pi\mathrm{ML}_{fin}$, but modulo the non-structural subtyping relation.

Conclusion

We have described a simple, flexible and efficient solution for extending ML with operations on records allowing some sort of inheritance. The solution uses an extension of ML with a sorted

[2] In [FM89], the structural subtyping is *open*. With open structural subtyping only some of the atomic coercions are known, but there are potentially many others that can be used (opened) during typechecking of later phrases of the program. Closed subtyping is usually easier than open subtyping.

$$\text{If } \alpha \in \mathcal{V}(\tau) \wedge \tau \in e \setminus \mathcal{V}, \qquad \frac{U \wedge (\alpha \mapsto \sigma)(e)}{U \wedge \exists\, \alpha \cdot (e \wedge \alpha = \sigma)} \qquad \text{(Generalize)}$$

$$\frac{U \wedge a : \tau \ ; \ \tau' = \mathrm{abs} \underset{\sim}{=} e}{U \wedge \bigwedge \begin{cases} \mathrm{abs} = e \\ \tau = \mathrm{abs} \\ \tau' = \mathrm{abs} \end{cases}} \qquad \frac{U \wedge a : \alpha \ ; \ \alpha' = b : \beta \ ; \ \beta' \underset{\sim}{=} e}{U \wedge \exists\, \gamma \cdot \bigwedge \begin{cases} b : \beta \ ; \ \beta' = e \\ \alpha' = b : \beta \ ; \ \gamma \\ \beta' = a : \alpha \ ; \ \gamma \end{cases}} \qquad \text{(Mutate)}$$

$$\frac{U \wedge f(\tau_1, \ldots \tau_p) = f(\alpha_1, \ldots \alpha_p) \underset{\sim}{=} e}{U \wedge \bigwedge \begin{cases} f(\alpha_1, \ldots \alpha_p) = e \\ \tau_i = \alpha_i, \qquad i \in [1, p] \end{cases}} \qquad \text{(Decompose)}$$

$$\frac{U \wedge \alpha = e \wedge \alpha \underset{\sim}{=} e'}{U \wedge \alpha = e = e'} \qquad \text{(Fuse)}$$

Figure 1: Rewriting rules for record-type unification

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. In many cases, the problem can be solved by inserting retyping functions. We also propose structural subtyping as a more systematic solution. But it is not clear yet whether we would want such an extension, for it might not be worth the extra cost in type inference.

Acknowledgments

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

A Unification on record types

The algorithm is an adaptation of the one given in [Rém92b], which we recommend for a more thorough presentation. It is described by transformations on unificands that keep unchanged the set of solutions. Multi-equations are multi-sets of terms, written $\tau_1 = \ldots \tau_p$, and unificands are systems of multi-equations, that is, multi-sets of multi-equations, with existential quantifiers. Systems of multi-equations are written U. The union of systems of multi-equations (as multi-sets) is written $U \wedge U'$ and $\exists\, \alpha \cdot U$ is the existential quantification of α in U. Indeed, $\exists$ acts as a binder and systems of multi-equations are taken modulo α-conversion, permutation of consecutive binders, and $\exists\, \alpha \cdot U$ is assumed equal to U whenever α is not free in U. We also consider both unificands $U \wedge \exists\, \alpha \cdot U'$ and $\exists\, \alpha \cdot U \wedge U'$ equal whenever α is not in U. Any unificand can be written $\exists\, W \cdot U$ where W is a set of variables, and U does not contain any existantial.

The algorithm reduces a unificand into a solved unificand in three steps, or fails. The first step is described by rewriting rules of figure 1. Rewriting always terminates. A unificand that cannot be transformed anymore is said completely decomposed if no multi-equation has more than one non-variable term, and the algorithm pursues with the occur check while instantiating the equations

by partial solutions as described below, otherwise the unificand is not solvable and the algorithm fails.

We say that a multi-equation e' is inner a multi-equation e if there is at least a variable term of e' that appears in a non-variable term of e, and we write $e' \lessdot e$. We also write $U' \not\lessdot U$ for

$$\forall e' \in U',\ \forall e \in U,\ e' \not\lessdot e$$

The system U is independent if $U \not\lessdot U$.

The second step applies the rule

$$\text{If } e \wedge U \not\lessdot e, \qquad \frac{e \wedge U}{e \wedge \hat{e}(U)} \qquad \text{(Replace)}$$

until all possible candidates e have fired the rule once, where $\hat{e}$ is the trivial solution of e that sends all variable terms to the non-variable term if it exists, or to any (but fixed) variable term otherwise. If the resulting system U is *independent* (i.e. $U \not\lessdot U$), then the algorithm pursues as described below; otherwise it fails and U is not solvable.

Last step eliminates useless existential quantifiers and singleton multi-equations by repeated application of the rules:

$$\text{If } \alpha \notin e \wedge U, \qquad \frac{\exists\, \alpha \cdot (\alpha = e \wedge U)}{e \wedge U} \qquad \frac{\{\tau\} \wedge U}{U} \qquad \text{(Garbage)}$$

This always succeeds, with a system $\exists\, W \cdot U$ that is still independent. A principal solution of the system is $\hat{U}$, that is, the composition, in any order, of the trivial solutions of its multi-equations. It is defined up to a renaming of variables in W. The soundness and correctness of this algorithm is described in [Rém92b].

The Replace step is actually not necessary, and a principal solution can be directly read from a completely decomposed form provided the transitive closure of the inner relation on the system is acyclic (see [Rém92b] for details).

With the signature $\mathcal{S}''$ the only change to the algorithm is the addition of the mutation rules:

$$\frac{a : \tau\ ;\ \tau' = pre\ = e}{\bigwedge \begin{cases} pre\ = e \\ \tau = pre \\ \tau' = pre \end{cases}} \qquad \frac{a : \alpha\ ;\ \beta = \gamma_1.\gamma_2 = e}{\exists\, \alpha_1\alpha_2\beta_1\beta_2 \cdot \bigwedge \begin{cases} \gamma_1.\gamma_2 = e \\ \alpha = \alpha_1.\alpha_2 \\ \beta = \beta_1.\beta_2 \\ \gamma_1 = a : \alpha_1.\beta_1 \\ \gamma_2 = a : \alpha_2.\beta_2 \end{cases}}$$

Note that in the first mutation rule, all occurrences of *pre* in the conclusion (the right hand side) of the rewriting rule have different sorts and the three equations could not be merged into a multi-equation. They surely will not be merged later since a common constant cannot fire fusion of two equations (only a variable can). As all rules are well sorted, rewriting keeps unificands well sorted.

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*, volume 173 of *Lecture Notes in Computer Science*, pages 51–68. Springer Verlag, 1984. Also in Information and Computation, 1988.

[Car86] Luca Cardelli. Amber. In *Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*, 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*, Lecture Notes in Computer Science. Springer Verlag, 1989.

[Car91] Luca Cardelli. Extensible records in a pure calculus of subtyping. Private Communication, 1991.

[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*. INRIA-Rocquencourt, BP 105, F-78 153 Le Chesnay Cedex, 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*, volume 88 of *Lecture Notes in Computer Science*, pages 194–204. Springer Verlag, 1980.

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

[FM88] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In *ESOP '88*, volume 300 of *Lecture Notes in Computer Science*, 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.

[HMT91] Robert Harper, Robin Milner, and Mads Tofte. *The definition of Standard ML*. The MIT Press, 1991.

[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* $1, 2, \ldots, \omega$. Thèse de doctorat d'état, 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*, volume 230. 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.

[Rém89] Didier Rémy. Records and variants as a natural extension of ML. In *Sixteenth Annual Symposium on Principles Of Programming Languages*, 1989.

[Rém90] Didier Rémy. *Algèbres Touffues. Application au Typage Polymorphe des Objects Enregistrements dans les Langages Fonctionnels.* Thèse de doctorat, Université de Paris 7, 1990.

[Rém92a] Didier Rémy. Extending ML type system with a sorted equational theory. Research report 1766, INRIA-Rocquencourt, BP 105, F-78 153 Le Chesnay Cedex, 1992.

[Rém92b] Didier Rémy. Syntactic theories and the algebra of record terms. Research report 1869, INRIA-Rocquencourt, BP 105, F-78 153 Le Chesnay Cedex, 1992.

[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.* BP 105, F-78 153 Le Chesnay Cedex, France, 1989.