INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1)39 63 55 11

Rapports de Recherche

N°1739

Programme 2

*Calcul symbolique, Programmation
et Génie logiciel*

TYPING RECORD CONCATENATION FOR FREE

Didier Rémy

Août 1992

Typing Record Concatenation for Free

Didier Rémy*

*This work was partly supported by research grant NSF IRI86-10617.

Typing Record Concatenation for Free

Abstract

We show that any functional language with record extension possesses record concatenation for free. We exhibit a translation from the latter into the former. We obtain a type system for a language with record concatenation by composing the translation with type-checking in a language with record extension. We apply this method to a version of ML with record extension and obtain an extension of ML with either asymmetric or symmetric concatenation. The latter extension is simple, flexible and has a very efficient type inference algorithm in practice. Concatenation together with removal of fields needs one more construct than extension of records. It can be added to the version of ML with record extension. However, many typed languages with record cannot type such a construct. The method still applies to them, producing type systems for record concatenation without removal of fields. Object systems also benefit of the encoding which shows that multiple inheritance does not actually require the concatenation of records but only their extension.

Le Typage de la Concatenation des Objets Enregistrements Gratuitement

Résumé

Nous montrons que tout langage fonctionnel qui possède des enregistrements avec une opération d'extension possède gratuitement une opération de concaténation. Nous exhibons une traduction de la seconde vers la première. Un système de types pour un langage possédant la concaténation des enregistrements peut être obtenu en composant la traduction avec le typage dans un langage ayant une opération d'extension des enregistrements. La méthode est appliquée à une version de ML avec une opération d'extension des enregistrements et permet d'obtenir une extension de ML avec une opération de concaténation symétrique ou assymétrique. Le langage obtenu est simple, souple et possède un algorithme de synthèse de type efficace en pratique. L'opération de concaténation simultanément avec le retrait de champs nécessitent une opération de plus que l'extension des enregistrements. Celle-ci peut être ajoutée à la version de ML ci-dessus. Cependant, la plupart des langages typés avec des enregistrements ne peuvent pas être facilement étendus avec une telle construction. La méthode est encore applicable, produisant des systèmes de typage pour la concaténation mais sans le retrait des champs. Les systèmes objets bénéficient également du codage, puisqu'il montre que l'héritage multiple ne nécessite pas la concaténation des enregistrements et mais seulement leur extension.

Introduction

Dictionaries are an imported data abstraction in functional programming languages. They are basically partial functions from keys to values. A simple implementation of dictionaries is the *association list*, commonly called *A-list* . A-lists are lists of pairs, the first component being the key to access the value of the second component. The usual *cons* and *append* operations provide facilities for extending the domain of an A-list and merging two A-lists into one defined on the union of the domains of the input lists, respectively. Access to a given key may fail when the key is not in the domain of the A-list, which cannot be checked statically. Records are a highly restricted form of A-lists. Keys may no longer be any values, but belong to a distinguished set of atomic values, called labels. All fields of a record must be specified at creation time. These restrictions make it possible to perform static checks on accesses to record fields.

Then, an important goal in typechecking records, was to allow a record with many fields to be used instead of a records with fewer fields. This was first suggested by Cardelli in the language Amber [Car86] using inclusion on monomorphic types.

Later, Wand [Wan87] used polymorphism instead of a specific inclusion relation on types. He also re-imported the *cons* operation of A-lists which became the extension of records with new fields. Originally, this construction was free (existing fields could be redefined), but strict versions (existing fields could not be redefined) have been proposed [OB88, JM88] to avoid typechecking difficulties. Note that *cons* on A-lists naturally implements free extension.

Record extension quickly became popular, but many languages still only provide the strict version [JM88, Oho90, HP90a]. Finally Wand re-imported the *append* of A-lists, calling it record concatenation. An important motivation for this is the encoding of multiple inheritance [Wan89] in object oriented languages.

Record concatenation is still considered a challenge, since it is either very restricted [HP90a] or leads to combinatorial explosion of typechecking [Wan88]. We propose a general approach to concatenation. In fact we claim that concatenation comes for free once record extension is provided. We justify this assertion by presenting an encoding of the latter into the former. The interest of the encoding is to provide a type system for record concatenation by composing the coding with a type system for record extension.

We introduce the translation in an untyped framework in section 1. In section 2, we apply it to an extension of ML for record extension. In the last section we briefly illustrate the encoding on a few other languages.

1 Encoding of concatenation

In this section we describe how concatenation can be encoded with extension. The language with record extension, L, is an extension of the untyped λ calculus plus distinguished constructs for record expressions:

$M ::= x$	variable
$\mid \lambda\, x.\, M$	abstraction
$\mid M\ M$	application
$\mid \{\}$	empty record
$\mid \{M \text{ with } a = M\}$	record extension
$\mid M.a$	record access

The semantics of records is the usual one. Informally, they are partial functions from labels to values. The empty record is defined nowhere. Accessing a field of a record is applying the record to that field. It produces an error if the accessed field is not defined. The *free* extension of a record with a new field defines or redefines that field with the new value. The *strict* extension does the same if the field was undefined, but produces an error otherwise. In an untyped language the free extension is preferred since the more well typed programs, the better.

The *concatenation* (or *merge*) operator $\|$ takes two records and returns a new record composed of all fields defined in any of its arguments. There are different semantics given to the merge, when both records define the same field: *symmetric* concatenation rejects this case [HP90b] while *asymmetric* concatenation takes the value from the last record [Wan89]. We will not consider *recursive* concatenation that would compute the concatenation of common fields by recursively concatenating their values.

The language with record concatenation, $L^{\|}$ is

$$M ::= x \;\Big|\; \lambda\, x.\, M \;\Big|\; M\ M \;\Big|\; \{\} \;\Big|\; \{a = M\} \;\Big|\; M \parallel M \;\Big|\; M.a$$

The language is an extension of L with a construct for concatenation, but record extension has been replaced by one-field records that are more primitive in the presence of concatenation, since[1]:

$$\{M \text{ with } a = N\} \equiv M \parallel \{a = N\}$$

Reading this equality from right to left is also interesting: it means that one-field concatenation can be written with record extension only. It gives the expected semantics of asymmetric concatenation when the extension is free and the semantics of symmetric concatenation when the extension is strict. We are going to generalize this to a translation from the language $L^{\|}$ to the language L.

1.1 The untyped translation

The following translation works for both asymmetric and symmetric concatenation. We arbitrarily choose asymmetric concatenation.

The extension of fields provides the one-field concatenation operation:

$$\lambda\, r.\, (r \parallel \{a = M\}) = \lambda\, r.\, \{r \text{ with } a = M\},$$

which we write $\{a = M\}^{\dagger}$. In fact, we can compute $r \parallel s$ whenever we know exactly the fields of s, since

$$r \parallel \{a_1 = M_1\ ;\ \ldots a_n = M_n\} \equiv \{\ldots\{r \text{ with } a_1 = M_1\} \ldots \text{ with } a_n = M_n\}.$$

This equivalence could also have been deduced from the decomposition of s into one-field concatenations

$$(\ldots(r \parallel \{a_1 = M_1\}) \ldots \parallel \{a_n = M_n\}),$$

which is also the composition

$$(\{a_n = M_n\}^{\dagger} \circ \ldots \{a_1 = M_1\}^{\dagger})\ r.$$

We write

$$\{a_1 = M_1\ ;\ \ldots a_n = M_n\}^{\dagger}$$

[1]This is similar to the correspondence between *append* and *cons* on A-lists, in this particular case, the equality is
$[M]$ *append* $r = M$ *cons* r.

for the abstraction of the previous expression over r. More generally we define the transformation $\dagger$ on record expressions, called *record abstraction*, by:

$$\begin{aligned} \{\}^\dagger &\equiv \lambda\, u.\, u \\ \{a = M\}^\dagger &\equiv \lambda\, u.\, \{u \text{ with } a = M^\dagger\} \\ (M \parallel N)^\dagger &\equiv N^\dagger \circ M^\dagger \end{aligned}$$

Since any record expression can be decomposed into a combination of the three previous forms, the transformation is defined for all records. It satisfies the property

$$r^\dagger = \lambda\, u.\, (u \parallel r).$$

Thus r is equal to $r^\dagger\ \{\}$. If we transform all record expressions in a program, then we have to replace the access $r.a$ by $(r^\dagger\ \{\}).a$. Actually, it is enough to apply r to a record r' that does not contain the a field and read the a field from the result $(r\ r').a$. In a typed language this solution will leave more flexibility for the type of r. Other constructs of the languages simply propagate the translation. Thus the translation is completed by

$$\begin{aligned} (M.a)^\dagger &\equiv (M^\dagger\{\}).a \\ (\lambda\, x.\, M)^\dagger &\equiv \lambda\, x.\, M^\dagger \\ (M\ N)^\dagger &\equiv M^\dagger\ N^\dagger \\ x^\dagger &\equiv x \end{aligned}$$

The translation works quite well in an untyped framework. However, the encoding is not injective, for instance it identifies the empty record with the identity function. In the next section we adapt the translation to a typed framework.

1.2 The tagged translation

In this section we improve the translation so that the encoding becomes injective. The main motivation is to prepare the use of the encoding to get a typed version of $L^{\parallel}$ by pulling back the typing rules of a typed version of L. The well typed programs of $L^{\parallel}$ will be the reverse image of the well typed programs of L. The translation should be injective on well typed programs. A solution is to tag the encoding of records, so that they become tagged abstractions, distinct from other abstractions.

In fact we replace L by $L^{\text{Tag},\text{Untag}}$, that is L plus two constants Tag and Untag used to tag and untag values. The only reduction involving Tag or Untag is that Untag (Tag M) reduces to M. Tag and Untag can be thought as the unique constructor and the unique destructor of an abstract data type, respectively. In SML [HMT91] they could be defined as:

```
abstype (α, β) tagged = Tagged of α → β with
  val Tag = fn x ⇒ Tagged x
  val Untag = fn Tagged x ⇒ x
end;
```

Their role is to certify that some functional values are in fact record abstractions, Tag stamps them and Untag reads and removes the stamps. Obviously, these constants are not accessible in $L^{\parallel}$, i.e. they are introduced during the translation only.

Syntactically the existence of Tag and Untag is not a question, but semantically a model of a calculus with record extension might not possess such constants. On the opposite, finding a particular model in which the constants Tag and Untag exists might be as difficult as finding a direct model for concatenation. Anyhow, we limit our use of the encoding to syntactic issues.

The tagged translation is:

$$\begin{aligned}
\{\}^\dagger &\equiv \mathrm{Tag}\,(\lambda\, u.\, u)\\
\{a = M\}^\dagger &\equiv \mathrm{Tag}\,(\lambda\, u.\, \{u \text{ with } a = M^\dagger\})\\
(M \parallel N)^\dagger &\equiv \mathrm{Tag}\,(\lambda\, u.\ \mathrm{Untag}\ N^\dagger\ (\mathrm{Untag}\, M^\dagger\ u)))\\
(M.a)^\dagger &\equiv ((\mathrm{Untag}\ M)\ \{\}).a
\end{aligned}$$

It does not modify other constructs:

$$\begin{aligned}
(\lambda\, x.\, M)^\dagger &\equiv \lambda\, x.\, M^\dagger\\
(M\ N)^\dagger &\equiv M^\dagger\ N^\dagger\\
x^\dagger &\equiv x
\end{aligned}$$

We would like to show a property such as: starting with a calculus of record extension, we can translate any program of a calculus with record concatenation into the first calculus enriched with constants Tag and Untag using the translation above, and thereby get — in some sense — an equivalent program.

$$L^{\parallel} \xrightarrow{\dagger} L^{\{\mathrm{Tag},\mathrm{Untag}\}}$$

$$\begin{tikzcd}
M^{\parallel} \arrow[r, mapsto] \arrow[d, "\mathrm{eval}"', twoheadrightarrow] & M \arrow[d, "\mathrm{eval}", twoheadrightarrow] \\
v^{\parallel} \arrow[r, dotted, no head, "?"] & v
\end{tikzcd}$$

Without any such result, the translation † is no more than a good intuition to understanding record concatenation. In the next section it helps finding a type system for a language with concatenation $L^{\parallel}$ from a typed language with extension L, by translating $L^{\parallel}$ programs and then typing them in L.

1.3 Concatenation with removal of fields

We omitted one construction in the language L: the restriction of fields. We extend both languages L and $L^{\parallel}$ with record *restriction*:

$$M ::= \ \ldots \mid M \setminus a$$

Record restriction takes a record and removes the corresponding field from its domain. As for extension of fields, restriction of fields can be free or strict. We consider free restriction here. The question is obviously the extension of the transformation † to restriction of fields.

The guide line is to keep the equality

$$(M \setminus a)^\dagger = \lambda\, u.\, u \parallel (M \setminus a)$$

true, since it was true before the introduction of restriction of fields. Actually this equality is needed since it is the basis of the translation of the extraction of fields.

Unfortunately, the attempt

$$(M \setminus a)^\dagger = \lambda\, u.\, (M^\dagger u) \setminus a$$

does not work: the record $u \parallel (M \setminus a)$ is not equal to $(M^\dagger\ u) \setminus a$, since if the record u provides an a field, this field is defined in the left expression but it is undefined in the right expression.

In fact $u \,\|\, (M \setminus a)$ is equal to $M^\dagger\, u$ on all fields but a. On the a field it is undefined if u is, or defined with the value of the a field of u otherwise. This operation cannot be written in the language L; we need another construct,

$$\{M \text{ but } a \text{ from } N\},$$

called *combining*. From two records M and N, it defines one that behaves exactly as M on all fields but a, and as N on the a field. This primitive is stronger than $(\_ \setminus a)$ which could be defined as $\{\_ \text{ but } a \text{ from } \{\}\}$.

Now, the translation of $(M \setminus a)$ can be defined by

$$(M \setminus a)^\dagger \equiv \lambda\, u.\, \{M^\dagger\, u \text{ but } a \text{ from } u\}$$

Its tagged version is:

$$(M \setminus a)^\dagger \equiv \mathrm{Tag}\,(\lambda\, .\, \{\mathrm{Untag}\,(M^\dagger\, u) \text{ but } a \text{ from } u\})$$

We call L^+ the language L extended with the combining construct. This construct has never been introduced in the literature before. If the language L is typed, it may be the case that the combining primitive cannot be assigned a correct and decent type in the type system of L and L^+ might not be a trivial extension of L or even not exist.

The combining construct is not in $L^{\|}$ and there is no easy way to provide it in an extension of $L^{\|}$. Therefore L but not L^+ is a sub-language of $L^{\|}$.

2 Application to a natural extension of ML

In this section we apply the translation where L is a version of ML with record extension, and we get a language with record concatenation. We first review the language Π taken from [Rém90, Rém92c] for record extension. Then we describe in detail two versions of the typed language $\Pi^{\|}$ obtained by pulling back the typing rules of Π. Last, we discuss the system $\Pi^{\|}$ on its own, and compare it with other existing systems with concatenation.

2.1 An extension of ML for records

The language, called Π, is taken from [Rém90, Rém92c]. It is an extension of ML, where the language of types has been enriched with record types in such a way that record operations can be introduced as primitive functions rather than built in constructs. The main properties are described in [Rém92c] and proved in [Rém90, Rém92b, Rém92a]. The following summary should be sufficient for understanding the next sections. The reader is referred to [Rém92c] for a more thorough presentation.

Let $\mathcal{L}$ be a finite set of labels. We write a, b and c for labels and L for finite subsets of labels. The language of types is informally described by the following grammar (a formal description using sorts can be found in [Rém92c]):

$\tau ::= \alpha \mid \tau \to \tau \mid \Pi\rho^{\emptyset}$		types
$\rho^L ::= \chi^L \mid \mathrm{abs}^L \mid a : \varphi \,;\, \rho^{L\cup\{a\}}$	$a \notin L$	rows defining all labels but those in L
$\varphi ::= \theta \mid \mathrm{abs} \mid \mathrm{pre}\,(\tau)$		fields

where α, β, γ and δ are type variables, χ, π and ξ are row variables and θ and ε are field variables.

Intuitively, a row with superscript L describes all fields but those in L, and tells for each of them whether it is present with a value of type τ (positive information $\mathrm{pre}\,(\tau)$) or absent (negative information abs). A *template* row is either abs or a row variable. It always describes an infinite set of fields. The superscripts in row expressions L are finite sets of labels. Their main role is to prevent fields from being defined twice: the type

$$\left(a : \theta \;;\; (a : \varepsilon \;;\; \chi^L)\right)$$

cannot be written for any L. Similarly, all occurrences of the same row variable should be preceded by the same set of labels (possibly in a different order). The type

$$\left(a : \theta \;;\; \chi^L\right) \to \left(\chi^L\right).$$

cannot be written either, since the row variable χ cannot be both in the syntactic class of rows not defining label a and the syntactic class of rows defining all labels. The superscripts are part of the syntax, but we shall omit them whenever they are obvious from context. We write $a : \alpha \;;\; b : \beta \;;\; \gamma$ for $a : \alpha \;;\; (b : \beta \;;\; \gamma)$.

Example 1 The following is a well-formed record type:

$$\alpha \to (a : \mathrm{pre}\,(\alpha) \;;\; b : \mathrm{pre}\,(\mathrm{num}) \;;\; \mathrm{abs}\,)$$

Types are equal modulo the following equations:

- left commutativity, to reorder fields:

$$(a : \theta \;;\; b : \varepsilon \;;\; \chi) = (b : \varepsilon \;;\; a : \theta \;;\; \chi)$$

- distributivity, to access absent fields:

$$\mathrm{abs} \;=\; (a : \mathrm{abs} \;;\; \mathrm{abs}\,)$$

Example 2 The record types $(a : \mathrm{pre}\,(\alpha) \;;\; \mathrm{abs}\,)$ and $(b : \mathrm{abs} \;;\; a : \mathrm{pre}\,(\alpha) \;;\; \mathrm{abs}\,)$ are equal.

Any field defined by a template can be extracted from it using substitution if the template is a variable or distributivity if it is abs.

Example 3 In $(a : \mathrm{pre}\,(\alpha) \;;\; \mathrm{abs}\,)$, the template is abs; its superscript is $\{a\}$. To read the b field, we replace abs by $b : \mathrm{abs} \;;\; \mathrm{abs}$. The original type becomes $(a : \mathrm{pre}\,(\alpha) \;;\; b : \mathrm{abs} \;;\; \mathrm{abs}\,)$, and the new template has superscript $\{a, b\}$.

In $(a : \mathrm{pre}\,(\alpha) \;;\; \chi)$, the χ variable can be substituted by $b : \varepsilon \;;\; \pi$. The type becomes

$$(a : \mathrm{pre}\,(\alpha) \;;\; b : \varepsilon \;;\; \pi)$$

and π is the new template.

The language of expressions is the core ML language.

$$M ::= x \mid c \mid \lambda\, x.\, M \mid M\ M \mid \mathrm{let}\ x = M\ \mathrm{in}\ M$$

where the constants c include the following primitives operating on records, with their types:

$$\begin{aligned} \{\} &: (\mathrm{abs}\,) \\ \_.a &: (a : \mathrm{pre}\,(\alpha) \;;\; \chi) \to \alpha \\ \{\_\ \mathrm{with}\ a = \_\} &: (a : \mathrm{abs} \;;\; \chi) \to \alpha \to (a : \mathrm{pre}\,(\alpha) \;;\; \chi) \\ \_ \setminus a &: (a : \theta \;;\; \chi) \to (a : \mathrm{abs} \;;\; \chi) \end{aligned}$$

Primitives for Record Extension (II)

The extension on a field $\{\_ \text{ with } a = \_\}$ is strict: a field can only be added to a record r that does not already possess this field. But the restriction of a field $\_ \setminus a$ is free: it can be applied to a record which does not have field a. Free extension with a field b is achieved by restriction of field b followed by strict extension with field b. That is, it is the composition:

$$(\_ \setminus a) \circ (\{\_ \text{ with } a = \_\}) : (a : \theta \; ; \chi) \; \to \alpha \to (a : \text{pre}\,(\alpha) \; ; \chi)$$

that we abbreviate $\{\_ \text{ with } !a = \_\}$. In the simplest language, the restriction of fields would not be provided, and the extension would be given whether strict or free.

Typing rules are the same as those of ML but where type equality is taken modulo the equations. As in ML, any typeable expression possesses a principal type. We show a few examples extracted from [Rém92c] and run on a CAML prototype.

Records are built all at once as in

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

or from previous records by removing or adding fields:

```
#let truck = {car \ age with name = "Blazer"; registration = 6587867567};;
truck:Π (name:pre (string); registration:pre (num); age:abs; abs)
```

Fields are accessed as usual with the "dot" operation.

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

Here, the field registration must be defined with a value of type $\boldsymbol{\alpha}$, so the field registration has type pre $(\boldsymbol{\alpha})$, and other fields may or may not be defined; they are grouped in the template variable $\boldsymbol{\chi}$. The return value has type $\boldsymbol{\alpha}$. The function eq below takes two records possessing at least a registration field of the same type[2]:

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

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

The identifier "it" is bound to the last toplevel phrase (the prototype types the expressions but it does not evaluate them). The two records car and truck do not have the same set of fields, but both can still be passed to the function registration.

2.2 An extension of ML with record concatenation

The language Π described in section 2.1 can easily be extended with a combining primitive

$$\{\_ \text{ but } a \text{ from } \_\} : (a : \theta \; ; \chi) \; \to (a : \varepsilon \; ; \pi) \; \to (a : \varepsilon \; ; \chi)$$

The extended language is referred to as Π^+. We apply the transformation $\dagger$ with Π as L. We first consider the strict version of Π^+, then we show a few examples and we treat the free version of Π^+ at the end.

[2]For simplicity of examples we assume the existence a polymorphic equality equal.

Symmetric concatenation

We encode the language $\Pi^{||}$ with symmetric concatenation into the version of Π^{+} with strict extension. We introduce a new type symbol $\{\,\_ \Rrightarrow \_\,\}$ of arity two, and we assume given the two constants:

$$\begin{aligned} \text{Tag} &: ((\chi) \to (\pi)) \to \{\, \chi \Rrightarrow \pi \,\}, \\ \text{Untag} &: \{\, \chi \Rrightarrow \pi \,\} \to ((\chi) \to (\pi)). \end{aligned}$$

They are private to the translation.

A program is typable in $\Pi^{||}$ if and only if its translation is typable in $\Pi^{\text{Tag},\text{Untag}}$ (Π extended with Tag and Untag). However, composing the translation with typechecking in $\Pi^{\text{Tag},\text{Untag}}$ is the same as typechecking in $\Pi^{||}$ with the following types for primitives:

$$\begin{aligned} \{\} &: \{\, \chi \Rrightarrow \chi \,\} \\ \_.a &: \{\, a : \text{abs}\,; \chi \Rrightarrow a : \text{pre}\,(\alpha); \pi \,\} \to \alpha \\ \{a = \_\} &: \alpha \to \{\, a : \text{abs}\,; \chi \Rrightarrow a : \text{pre}\,(\alpha); \chi \,\} \\ \backslash a &: \{\, a : \theta; \chi \Rrightarrow a : \varepsilon; \pi \,\} \to \{\, a : \theta'; \chi \Rrightarrow a : \theta'; \pi \,\} \\ || &: \{\, \chi \Rrightarrow \pi \,\} \to \{\, \pi \Rrightarrow \xi \,\} \to \{\, \chi \Rrightarrow \xi \,\} \end{aligned}$$

Primitives for symmetric concatenation ($\Pi^{||}$)

Thus the translation can be avoided.

When typing directly in $\Pi^{||}$ with the rules above, all record types are written with $\{\,\_ \Rrightarrow \_\,\}$ and the type symbol Π can be removed; the grammar for types becomes

$$\tau ::= \alpha \mid \tau \to \tau \mid \{\rho^{\emptyset} \Rrightarrow \rho^{\emptyset}\}$$

The type $\{\, \chi \Rrightarrow \pi \,\}$ should be read "I am a record which given any input row of fields χ returns the output row π." The types for the primitives above can be read with the following intuition:

- The empty record returns the input row unchanged.
- As remarked above (section 1), we encoded the extraction of field a in M as the extraction of field a in the application of M to any record that does not contain the a field. Otherwise we would have got the weaker type:

  $$\_.a : \{\, \text{abs} \Rrightarrow a : \text{pre}\,(\alpha); \chi \,\} \to \alpha$$

  Thus, the extraction of the a field of r takes a record r which, given any row where a is absent, produces a row where a is defined with some value v. The result $r.a$ is this value v.
- A one-field record extends the input row, defining one more field (that should not be previously defined).
- The removal of field a from a record M returns a record that acts as M except on the field a where it acts as the empty record.
- Finally, concatenation composes its arguments.

It is easy to see that any program in Π is also a program in $\Pi^{||}$. First, define the extension primitive by:

$$\{M \text{ with } a = N\} \equiv M \mid\mid \{a = N\}$$

It has type:

$$\{\, \chi \Rrightarrow a : \text{abs}\,; \pi \,\} \to \alpha \to \{\, \chi \Rrightarrow a : \text{pre}\,(\alpha); \pi \,\}$$

Check that all the following typing assertions are correct in $\Pi^{||}$:

$$\begin{aligned}
\{\} &: \{\,\mathrm{abs} \Rightarrow \chi\,\} \\
\_.a &: \{\,\mathrm{abs} \Rightarrow a : \mathrm{pre}\,(\alpha); \pi\,\} \to \alpha \\
\{\_\ \mathrm{with}\ a = \_\} &: \{\,\mathrm{abs} \Rightarrow a : \mathrm{abs}\,; \pi\,\} \to \alpha \to \{\,\mathrm{abs} \Rightarrow a : \mathrm{pre}\,(\alpha); \pi\,\} \\
\_ \setminus a &: \{\,\mathrm{abs} \Rightarrow a : \varepsilon ; \pi\,\} \to \{\,\mathrm{abs} \Rightarrow a : \mathrm{abs}\,; \pi\,\}
\end{aligned}$$

Last, abbreviate $\{\,\mathrm{abs} : \rho\,\}$ as (ρ) to conclude that $\Pi^{||}$ possesses all the primitives of Π with all types that Π can assign to them. The rest of the language Π is core ML and is also in $\Pi^{||}$.

Examples

We show a few examples processed by a prototype written in CAML [CH89, Wei89]. The type inference engine is exactly the one of Π; only the primitives have changed. The syntax is similar to CAML syntax.

The type of a one-field record says that the record cannot be merged with another record that also defines this field:

```
#let a = {a = 1};;
a:{a:abs; χ ⇒ a:pre (num); χ}
```

Two records r and s can be merged if they do not define common fields. For instance, r can be merged on the left with {a = 1} if its output row on a is absent.

```
#let left r = r || {a = 1};;
left:{χ ⇒ a:abs; π} → {χ ⇒ a:pre (num); π}
```

The resulting record modifies its input row as r but on field a which is added. Similarly, s can be merged on the right with a if the input field a is present (with the adequate type).

```
#let right s = {a = 1} || s;;
right:{a:pre (num); χ ⇒ π} → {a:abs; χ ⇒ π}
```

In particular, s cannot define an a field, otherwise its input field a would be absent.

Non overwriting of fields is guaranteed on the left by negative information (absent field) at a positive row occurrence, and on the right by positive information (present field) at a positive row occurrence. Some symmetry is preserved! However writing r || s instead of s || r in a program sometime matters: one might typecheck while the other does not, though none of the programs would overwrite fields. If both typecheck, the type of the result will be the same (provided all fields are symmetric).

Here are a few more examples:

```
#let foo = fun r s → (r || s).a;;
foo:{a:abs; χ ⇒ π} → {π ⇒ a:pre (α); ξ} → α
```

This shows the functionality of concatenation on both sides. The result shall have an a field, but what argument will provide it is not specified yet.

```
#let gee = foo {b = 1};;
gee:{b:pre (num); a:abs; χ ⇒ a:pre (α); π} → α
```

Now r must define the a field.

```
#gee a;;
it:num
```

Asymmetric concatenation

The system Π may also provide free extension, with the following primitive:

$$\{\_ \text{ with } !a = \_\} : (a : \theta \ ; \chi) \to \alpha \to (a : \text{pre}\,(\alpha) \ ; \chi)$$

This will make concatenation asymmetric:

$$\{!a = \_\} : \alpha \to \{\, a : \theta ; \chi \Rightarrow a : \text{pre}\,(\alpha) ; \chi \,\}$$

For instance, the following example is typeable:

```
#let ab = (fun r → {!a = 1} || r) {!a = true; !b = 1};;
ab:{a:χ; b:π; ξ ⇒ a:pre (bool); b:pre (num); ξ}
```

This shows that asymmetric fields can be redefined with values of possibly incompatible types.

The choice between strict and free extension is encoded in the extension primitive, but the choice between asymmetric and symmetric concatenation is not encoded in the concatenation primitive which is always the composition. It is not concatenation which is symmetric or not, but record fields themselves! We can have symmetric and asymmetric fields coexisting peacefully.

```
#{!a = 1; b = true};;
it:{b:abs; a:χ; π ⇒ a:pre (num); b:pre (bool); π}
```

Primitives to modify these properties of fields can easily be provided

$$\begin{aligned} \text{symmetric}^a &: \{\, a : \theta ; \chi \Rightarrow a : \text{pre}\,(\alpha) ; \chi \,\} \to \{\, a : \text{abs}\,; \chi \Rightarrow a : \text{pre}\,(\alpha) ; \chi \,\} \\ \text{asymmetric}^a &: \{\, a : \theta ; \chi \Rightarrow a : \text{pre}\,(\alpha) ; \chi \,\} \to \{\, a : \varepsilon ; \chi \Rightarrow a : \text{pre}\,(\alpha) ; \chi \,\} \end{aligned}$$

But it is not possible to make all fields of a record symmetric, or asymmetric; this has to be done field by field.

We can now better understand why symmetric concatenation is not so symmetric. Both left and right functions accept any argument, and one should not expect them to behave the same on a record of which some of the fields are asymmetric.

With asymmetric fields, the following examples reach the limit of ML polymorphism. For instance, the function

```
#fun r s → s.b, r || s;;
it:{χ ⇒ b:abs; π} → {b:abs; π ⇒ b:pre (α); ξ} → α * {χ ⇒ b:pre (α); ξ}
```

does not accept a record r which has a b field, though the program would still run correctly if the b field of s is asymmetric. This is due to ML polymorphism weakness: the second argument is λ-bound and thus it is not polymorphic. The field b of s is observed by setting its input to abs, which has to be the output field b of r in r || s.

Since s has definitely a b field, the concatenation r || s is equal to the concatenation r\b || s. We can rewrite the previous program as

```
#fun r s → s.b, (r\b || s);;
it:{b:χ; π ⇒ b:ξ; φ} → {b:abs; φ ⇒ b:pre (α); ψ} → α * {b:abs; π ⇒ b:pre (α); ψ}
```

which can now be applied to any record r.

The restriction _\b of field b only changes the type of its arguments but does not modify it; it is called a *retyping function*. Many weaknesses of $\Pi^{||}$ originating in the restricted polymorphism provided by the ML type system can be solved by adding retyping functions. They insert type information in the program helping the type inference engine. We will describe other ways of solving these examples by strengthening the type inference engine in section 2.3.

2.3 Strength and weakness of $\Pi^{||}$

We compare our language with Wand's proposal [Wan89], and Harper and Pierce's system and mention possible extensions.

Comparison with other systems

There are only a few other systems that implement concatenation. Wand's proposal [Wan89] is still more powerful that our system $\Pi^{||}$. For instance

$$\lambda\, r.\; r.a + (\{\, a : 1 \,\} \;||\; r).a$$

is typable in Wand's system but not in ours. Wand's system polymorphism is carried by the concatenation operator, at the cost of bringing in the type system a restricted form of conjunctive types and having disjunction of principal types instead of unique principal types. In contrast, in our system, polymorphism is carried by records themselves. As mentioned above, we can regenerate polymorphism of records by inserting retyping functions. If the same restricted form of conjunctive types was brought in our system, then retyping functions would be powerful enough to regenerate all fields of a record without having to mention them explicitly. This would give back all the power of Wand's system.

This shows that the additional power of Wand's system comes from conjunctive types. Conversely, our system succeeds with only generic polymorphism on examples that needed conjunctive types in Wand's system. We are going to explain how this happens.

Wand's system can be reformulated in system Π. A simple idea is to type the concatenation operator by introducing an infix type operator $||$ of arity two:

$$||\;\; : (\chi) \to (\pi) \to (\chi \;||\; \pi)$$

But we have to eliminate $||$ operators that might hide type collisions. In the system Π, we entice distributing concatenation on fields with the equations:

$$(a : \theta \;;\; \chi) \;||\; (a : \varepsilon \;;\; \pi) = (a : \theta \;||\; \varepsilon \;;\; \chi \;||\; \pi)$$

The operator $||$ on fields can be defined by enumerating the triples $(\theta, \varepsilon, \theta \;||\; \varepsilon)$. They are all triples of the form

$$(\theta, \mathrm{abs}\,, \theta) \quad \text{or} \quad (\theta, \mathrm{pre}\,(\beta), \mathrm{pre}\,(\beta)).$$

This disjunction in the relation $||$ breaks the principal type property of type inference. Worse, disjunctions on different fields combine and make the resulting type (conjunction of types) explode in size.

Our system emphasizes that $\theta \;||\; \varepsilon$ is uniform on θ: once we know ε, we can eliminate the conjunction in $\theta \;||\; \varepsilon$. A field a, instead of carrying its type ε, carries the function $\theta \Rightarrow \theta \;||\; \varepsilon$. For instance, if M has type τ, the record $\{a = M\}$ would have type $(a : \mathrm{pre}\,(\tau) \;;\; \mathrm{abs}\,)$ in Π. On field a, since ε is now $\mathrm{pre}\,(\tau)$, the merging $\theta \;||\; \varepsilon$ is equal to $\mathrm{pre}\,(\tau)$. In the template, π is $\mathrm{abs}\,$, and thus $\chi \;||\; \pi$ is χ. We deduce the type in $\Pi^{||}$:

$$\{\, (a : \theta \Rightarrow \mathrm{pre}\,(\tau)); (\chi \Rightarrow \chi) \,\} \qquad \text{i.e.} \qquad \{\, (a : \theta; \chi) \Rightarrow (a : \mathrm{pre}\,(\tau); \chi) \,\}.$$

Another system with type inference was proposed by Ohori and Buneman in [OB88]. Their concatenation on records is recursive concatenation, which we do not provide. Note that they have a very restricted form of recursive concatenation since types in record fields must not contain any function type.

In explicitly typed languages, the only system with concatenation is the one of Harper and Pierce [HP90b]; it implements symmetric concatenation. Since their system is explicitly (higher order) typed, we say that typing a $\Pi^{||}$ program M succeeds in HP90 if we can find a HP90 program whose erasure (the program obtained by erasing all type information) is M. Their system has not free restriction of fields, but we shall ignore this difference.

The following $\Pi^{||}$ program cannot be typed in HP90:

```
#let either r s = (r || s).a in
#  if true then either {a = 1} {b = 2} else either {b = 2} {a = 1};;
it:num
```

In the expression (r || s).a, one has to choose whether r or s is defining field a, and thus the function either cannot be used with the two alternatives. This breaks the symmetry of concatenation.

Conversely, there are programs that can be typed in HP90 but not in $\Pi^{||}$ as a result of ML polymorphism restrictions. For instance the function

```
#let reverse r s = if true then r || s else s || r;;
```
reverse:$\{\chi \Rightarrow \chi\} \rightarrow \{\chi \Rightarrow \chi\} \rightarrow \{\chi \Rightarrow \chi\}$

cannot be applied to {a = 1} and {b = 2} in $\Pi^{||}$. In HP90 it would have type

$$\forall \chi \cdot \forall \pi \# \chi \cdot \chi \rightarrow \pi \rightarrow (\chi \parallel \pi)$$

and could be applied to any two compatible records. It is difficult, though, to tell whether the failure comes from a limitation of polymorphism in general, or the inability to quantify with constraints, since the two are strongly related. The typability of the previous example in $\Pi^{||}$ is somehow equivalent to the typability, in core ML, of the function:

```
#let reverse r s = if true then r o s else s o r;;
```
reverse:$(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

This is too weak a type! Whether a higher order language would give it a much better type is not so obvious. Next section provides a better basis for comparison between the two systems.

Limitations and extensions

Since the type inference engine of $\Pi^{||}$ is the same as the one of Π (only types of primitives have changed), both systems enjoy the same properties. Record polymorphism is provided by ML genericity introduced in let bindings. If this is too restrictive, then one should introduce type inclusion. One could also have a restricted conjunctive engine as in [Wan89]; however this would decrease considerably the efficiency of type inference, and the readability of types. Allowing recursion on types would also require an extension of the results (though in practice the mechanism is already present). In $\Pi^{||}$, as in Π, present fields cannot be implicitly forgotten, but have to be explicitly removed, unless the structure of fields is enriched with flags. All these improvements are discussed in detail in [Rém92c].

3 Other applications

The transformation can also be applied to other languages, which we illustrate in this section.

3.1 Application to Harper and Pierce's calculus.

The higher order typed language of Harper and Pierce [HP90a] already possesses concatenation, but records are not abstractions. It can still benefit from the encoding. Instead of presenting special constructs for operations on records, we could assume given the following primitives in their language:

$$\begin{aligned}
\{\} &: \{\,\} \\
\_.a &: \forall\,\alpha \cdot \forall\,\chi\#a \cdot (\{\,a:\alpha\,\} \parallel \chi) \to \alpha \\
\{a = \_\} &: \forall\,\alpha \cdot \alpha \to \{\,a:\alpha\,\} \\
\backslash a &: \forall\,\alpha \cdot \forall\,\chi\#a \cdot (\{\,a:\alpha\,\} \parallel \chi) \to \chi \\
\parallel\ &: \forall\,\chi \cdot \forall\,\pi\#\chi \cdot \chi \to \pi \to (\chi \parallel \pi)
\end{aligned}$$

Primitives for HP90

But the type system is not enough sophisticated to type the primitive {_ but a from _}. Thus we apply the translation dropping the removal of fields. Using the encoding, the primitive operations on records in the language HP90$^{\parallel}$ have the following types:

$$\begin{aligned}
\{\} &: \forall\,\chi \cdot (\chi \Rightarrow \chi) \\
\_.a &: \forall\,\alpha \cdot \forall\,\chi\#a \cdot \forall\,\pi\#a \cdot (\chi \Rightarrow \{\,a:\alpha\,\} \parallel \pi) \to \alpha \\
\{a = \_\} &: \forall\,\alpha \cdot \alpha \to \forall\,\chi\#a \cdot (\chi \Rightarrow \{\,a:\alpha\,\} \parallel \chi) \\
\parallel\ &: \forall\,\chi \cdot \forall\,\pi \cdot \forall\,\xi \cdot (\chi \Rightarrow \pi) \to (\pi \Rightarrow \xi) \to (\chi \Rightarrow \xi)
\end{aligned}$$

Primitives for HP90$^{\parallel}$

We can define a function either:

$$\begin{gathered}
\Lambda\,\alpha.\ \Lambda\,\chi.\ \Lambda\,\pi\#a.\ \lambda\,r : (\{\,\} \Rightarrow \chi).\ \lambda\,s : (\chi \Rightarrow \{\,a:\alpha\,\} \parallel \pi). \\
(\_.a\,[\alpha]\,[\{\,\}]\,[\pi]\ \ (\parallel\ \ [\{\,\}]\,[\chi]\,[\{\,a:\alpha\,\} \parallel \pi]\ \ r\ \ s))
\end{gathered}$$

and apply it to records {a = 1} and {b = 2} in any order. For instance,

$$\text{either}\ [\text{num}]\ [\{\,a:\text{num}\,\}]\ [\{\,b:\text{num}\,\}]\ \ (\{a = 1\}\ [\{\,\}])\ \ (\{b = 2\}\ [\{\,a:\text{num}\,\}])$$

Remind that this example is not typable in HP90.

Conversely, the program:

```
let reverse r s = if true then r || s else s || r in reverse {} {a = 1}, reverse {} {a = 1};;
```

can be typed in HP90, but we conjecture that it cannot be typed in HP90$^{\parallel}$. In fact its typability in HP90$^{\parallel}$ is equivalent to the following term being the erasure of a term of F:

```
(fun r → K (r I K) (r K I))
        (fun f g → (fun x → f (g x)) or (fun x → g (f x)))
```

where I is fun x → x and K is fun x y → x, and or is a constant assumed of type $\Lambda\,\alpha.\ \alpha \to \alpha \to \alpha$ in F.

To summarize, none of the language HP90 or HP90$^{\parallel}$ would be more powerful than the other. Remark that type applications and type abstractions are located in completely different places, thus a partial translation of explicitly typed terms from HP90$^{\parallel}$ to HP90 can only be global.

A previous language proposed by Harper and Pierce in [HP90a] had no concatenation, but shared the same spirit as HP90. The transformation applies to it as well, and results in a language with concatenation very closed to HP90$^{\parallel}$.

3.2 Application to Cardelli and Mitchell's calculus.

Unlike HP90, the language of Cardelli and Mitchell [CM89] does not already provide concatenation of records, but only strict extension. The application of our encoding to CM89 is not harder than to HP90. The language cannot be easily extended with the combining construct, therefore we skip the removal of fields. Using CM89 types, primitives for records operations in CM89$^{\|}$ have the following types:

$$\begin{aligned}
\{\} &: \forall\,\chi \cdot (\chi \Rightarrow \chi)\\
\_.a &: \forall\,\alpha \cdot \forall\,\chi < \langle\!\langle\rangle\!\rangle \setminus a \cdot \forall\,\pi < \langle\!\langle\rangle\!\rangle \setminus a \cdot (\chi \Rightarrow \langle\!\langle \pi \parallel a : \alpha \rangle\!\rangle) \rightarrow \alpha\\
\{a = \_\} &: \forall\,\alpha \cdot \alpha \rightarrow \forall\,\chi < \langle\!\langle\rangle\!\rangle \setminus a \cdot (\chi \Rightarrow \langle\!\langle \chi \parallel a : \alpha \rangle\!\rangle)\\
\parallel &: \forall\,\chi \cdot \forall\,\pi \cdot \forall\,\xi \cdot (\chi \Rightarrow \pi) \rightarrow (\pi \Rightarrow \xi) \rightarrow (\chi \Rightarrow \xi)
\end{aligned}$$

Primitives for CM89$^{\|}$

We can again define the function either:

$$\begin{aligned}
&\Lambda\,\alpha.\ \Lambda\,\chi.\ \Lambda\,\pi < \langle\!\langle\rangle\!\rangle \setminus a.\ \lambda\,r : (\langle\!\langle\rangle\!\rangle \Rightarrow \chi).\ \lambda\,s : (\chi \Rightarrow \langle\!\langle \pi \parallel a : \alpha \rangle\!\rangle).\\
&\quad (\_.a\ [\alpha]\ [\langle\!\langle\rangle\!\rangle]\ [\pi]\ \ (\parallel\ \ [\langle\!\langle\rangle\!\rangle]\ [\chi]\ [\langle\!\langle \pi \parallel a : \alpha \rangle\!\rangle]\ \ r\ \ s))
\end{aligned}$$

and apply it to the records $\{\mathsf{a} = 1\}$ and $\{\mathsf{b} = 2\}$:

$$\text{either}\ [\text{num}]\ [\langle\!\langle a : \text{num} \rangle\!\rangle]\ [\langle\!\langle b : \text{num} \rangle\!\rangle]\ \ (\{a = 1\}\ [\langle\!\langle\rangle\!\rangle])\ \ (\{b = 2\}\ [\langle\!\langle a : \text{num} \rangle\!\rangle])$$

3.3 Multiple inheritance without record concatenation

Multiple inheritance has been encoded with record concatenation [Wan89]. We have encoded record concatenation with record extension. By composition, multiple inheritance can be encoded with record extension.

Given the strengthening of the type inference engine to recursive types, the system $\Pi^{\|}$ would support multiple inheritance as presented in [Wan89]. But multiple inheritance makes very little use of concatenation. It is only necessary for building new methods, but objects do not need it. Thus it may be worth revisiting the typechecking of multiple inheritance of [Wan89] and eliminating the need for concatenation by abstracting methods as we abstracted records.

The following encoding of multiple inheritance was used by Wand in [Wan89]. The definition of a class

$$\text{class}\ (\vec{x})\ \text{inherits}\ \overrightarrow{P(\vec{Q})}\text{methods}\ \overrightarrow{a = M}\ \text{end}$$

was encoded as

$$\lambda\,\vec{x}.\ \lambda\,self.\ \overrightarrow{P(\vec{Q})} \parallel \{\overrightarrow{a = M}\}$$

The creation of objects of that class

$$\text{instance}\ C(\vec{N})$$

was the recursive expression

$$Y(C(\vec{N}))$$

Sending a method a to an object x was the same as reading the field x of a. The problem with this encoding is that it requires record concatenation. We can easily get read of it, using our trick. We encode a class definition as

$$\lambda\,\vec{x}.\ \lambda\,u.\ \lambda\,self.\ u \parallel \overrightarrow{P(\vec{Q})} \parallel \{\overrightarrow{a = M}\}$$

i.e.

$$\lambda\,\vec{x}.\ \lambda\,u.\ \lambda\,self.\ \{\overrightarrow{P(\vec{Q})}\ \circ u\ \text{with}\ \overrightarrow{a = M}\}$$

which only requires record extension. Then creating an object of that class becomes

$$Y(C(\vec{N})\{\})$$

and sending a method is unchanged.

Remarks

Since removing of fields is not needed here, this section applies to all typed calculi with record extension.

This section uses Wand's conception of inheritance. Objects are carrying their dictionarries. Other views of objets do not encode with record operations. This section does not apply to them.

Conclusion

We have described how a functional language with records and record extension automatically provides record concatenation. Though records are data, they should be typed as if there were abstractions over an input row of fields that they modify. Their behavior can be observed at any time by giving them the empty row as input. Concatenation is then composition.

We have applied the method to a record extension of ML. We have obtained a language implementing all operations on records except the recursive merge, allowing type inference in a very efficient way in practice.

The kind of type system that we have obtained seems complementary to Harper and Pierce's one. Taking the best of the two systems would be interesting investigation.

The encoding also helps understanding concatenation. However, the relationship between the semantics of a program in the language with concatenation and the semantics of its translation need to be investigated closely before claiming that concatenation itself comes for free.

Acknowledgments

I am grateful for interesting discussions with Luca Cardelli, Georges Gonthier, Jean-Jacques Lévy, Benjamin Pierce and Mitchell Wand, and particularly thankful to Xavier Leroy whose comments on the presentation of this article were very helpful.

References

[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.

[CH89] Guy Cousineau and Gérard Huet. *The CAML Primer.* 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.

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

[HP90a] Robert W. Harper and Benjamin C. Pierce. Extensible records without subsumption. Technical Report CMU-CS-90-102, Carnegie Mellon University, Pittsburg, Pensylvania, February 1990.

[HP90b] Robert W. Harper and Benjamin C. Pierce. A record calculus based on symmetric concatenation. Technical Report CMU-CS-90-157, Carnegie Mellon University, Pittsburg, Pensylvania, February 1990.

[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.

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

[Oho90] Atsushi Ohori. Extending ML polymorphism to record structure. Technical report, University of Glasgow, 1990.

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

[Rém91] Didier Rémy. Type inference for records in a natural extension of ML. Technical Report 1431, Inria-Rocquencourt, May 1991.

[Rém92a] Didier Rémy. Extending ML type system with a sorted equational theory. Technical report, BP 105, F-78 153 Le Chesnay Cedex, BP 105, F-78 153 Le Chesnay Cedex, 1992. To appear. Also in [Rém90], chapter 3.

[Rém92b] Didier Rémy. Syntactic theories and the algebra of record terms. Technical report, BP 105, F-78 153 Le Chesnay Cedex, BP 105, F-78 153 Le Chesnay Cedex, 1992. To appear. Also in [Rém90], chapter 2.

[Rém92c] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design.* MIT Press, 1992. To appear. Also in [Rém91].

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

[Wan88] Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *Third Symposium on Logic In Computer Science*, 1988.

[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.