Typechecking records and variants in a natural extension of ML

Didier Rémy

INRIA
Rocquencourt
France

Abstract

Strongly typed languages with records may have inclusion rules so that records with more fields can be used instead of records with less fields. But these rules lead to a global treatment of record types as a special case. We solve this problem by giving an ordinary status to records without any *ad hoc* assertions, replacing inclusion rules by extra information in record types. With this encoding ML naturally extends its polymorphism to records but any other host language will also transmit its power.

Introduction

Strongly typed languages gain a lot in practice by being polymorphic. They would also gain by allowing type inclusion, and L. Cardelli preferred inclusion to polymorphism in Amber [Car86]. For a long time it was not known how to mix the two notions.

J. Mitchell introduced type containment in [Mit84] in order to allow structural subtyping. He gave both a checking algorithm (C^2S) [1] for a second order type system and an inference algorithm (IS) for the first order case. Independently, L. Cardelli introduced a type system with records but no polymorphism [Car84], and gave a type checking algorithm ($C^0R^{\infty}_{\infty}$). He implemented a restriction of this system in Amber [Car86]. L. Cardelli and P. Wegner proposed an extension of this system in [CW85] to a second order language introducing the notion of bounded quantification, and gave a checking algorithm ($C^2R^{\infty}_{\infty}$).

M. Wand first tried to encode record inclusion with polymorphism but the original system [Wand87] was incomplete. He introduced a *with* construction for records [2] untypable by previous systems, but he has a less powerful inclusion on records which we call semi-inclusion, since he cannot forget fields from records but has just inclusion between functions accessing them. The inclusion is encoded with polymorphism and is thus limited to the depth of quantification, i.e. one. The whole system of M. Wand will be called (IR^1_0W). He presented a revised version [Wand88], but here the typing algorithm is also exponential relatively to the number of *with*.

R. Stansifer gave an inference algorithm ($I^-R^{\infty}_{\infty}$) for Cardelli's system, but his principal types may be empty since he does not check the consistency. Y-C. Fuh and P. Mishra introduced a general notion of subtyping (I^-G) [FM88]. In fact they essentially applied their system to the case of structural inclusion (IS). L. Jategaontar and J. Mitchell presented a new system [JM88] which mixes both structural subtyping (IS) and a restriction of Wand's system ($IR^1_0W^-$) which is complete and not exponential relatively to the number of *with*.

The encoding of inclusion presented here is a (IR^1_1W) system. As Wand's system it is based on polymorphism, but it also codes inclusion between records provided they have consistent fields, and it keeps the full power of Wand's *with* construction. It is based on a re-understanding of records, independent of any choice of the inclusion mechanism. With a simple (I) system, we get (IR^1_1W). Replacing (I)

[1] C^k stands for Checking at order k, I for Inference, S for Structural subtyping, R^p_q for Record subtyping where p and q respectively measure the level and the power of inclusion, and P^- indicates a restriction of the property P. Variants, *let* polymorphism and recursive types are not taken into account.

[2] noted (W)

Year	Author	Subtyping	Records	With construction
1984	L. Cardelli		$C^0R^{\infty}_{\infty}$	
1984	J. Mitchell	IS, C^2S		
1985	L. Cardelli & P. Wegner		$C^2R^{\infty}_{\infty}$	
1987	M. Wand			IR^1_0W
1988	R. Stansifer		$I^-R^{\infty}_{\infty}$	
1988	Y-C. Fuh & P. Mishra	I^-G, IS		
1988	L. Jategaontar & J. Mitchell	IS		$IR^1_0W^-$
				IR^1_1W coded in I
	In this paper			IR^{∞}_1W coded in IS
				$IR^{\infty}_{\infty}W$ coded in IS^+

Figure 1: Classification of systems with inclusion

by (IS) we get a system (IR^{∞}_1W) where inclusion is coded at any level, but we still need an extension (IS^+) of (IS) to non atomic inclusion relations to get the final system $(IR^{\infty}_{\infty}W)$ in order to be able to forget arbitrary fields. It generalizes Stansifer's system including the *with* construction and checking the consistency.

Most of the paper presents the encoding of inclusion with polymorphism, which is a solution to Wand's attempt but also a true extension since it includes polymorphic and recursive types and is more flexible. The main idea is to understand Wand's row variables as abbreviation schemes for field projections. Indeed, inferring types of variants has much more interest if we no longer need concrete type declarations.This justifies the introduction of recursion using unification on regular trees to infer more accurate types.

First we study the inclusion mechanism in a very simple language. Then we formally present our system and show how the usual inference algorithm is still applicable. Together with examples run on a CAML implementation [CAMLp][CAMLr], we compare our system with previous ones, discover its flexibility but also its limits in the encoding of inclusion. Finally we apply the method to systems with subtypes and recover the full power of inclusion.

1 An intuitive approach

Given a denumerable set of labels, records are partial finite functions from labels to values. We study the inclusion mechanism in a simple case:

- The set of labels is finite. Moreover we suppose there are only two labels. To help the intuition, we can represent partial functions by their graphs in two-field-boxes.
- We suppose there is a unique value $\bullet$ of type Δ. (read **black** of type **up**).

Without inclusion, we can write for instance:

$$X = \boxed{\bullet}\boxed{} \qquad Y = \boxed{\bullet}\boxed{\bullet}$$

but X and Y can neither be mixed, for example in the two branches of an *if* ... *then* ... *else* ... construct, nor can they be passed as arguments to the same function. With inclusion we would say that X and Y should represent the set of values:

$$X = \boxed{}\boxed{} \mid \boxed{\bullet}\boxed{}$$

$$Y = \boxed{}\boxed{} \mid \boxed{\bullet}\boxed{} \mid \boxed{}\boxed{\bullet} \mid \boxed{\bullet}\boxed{\bullet}$$

This agrees with the intuition that a record with more fields can be used instead of a record with less fields hiding some of them. But we do not want multiple values. So we give an ordinary status to empty fields, filling them with a new basic constant $\circ$ of type ∇ (read **white** of type **down**), which tells explicitly that the second projection does not make sense on the record X. So:

$$X = \boxed{\bullet}\boxed{\circ} \qquad Y = \boxed{\bullet}\boxed{\bullet}$$

We could type X and Y with

$$X : \Pi(\Delta, \nabla) \qquad Y : \Pi(\Delta, \Delta)$$

but Y could not be used instead of X. Thus we need that $\bullet$ has also type ∇. This is not a real trouble, we just assert that the basic constant $\bullet$ has both types Δ and ∇, i.e. it has principal type ε, where ε ranges only over the two types Δ and ∇. We have:

$$X : \Pi(\varepsilon, \nabla) \qquad Y : \Pi(\varepsilon, \varepsilon')$$

To access a component we must guarantee that it makes sense, i.e. that its value is not $\circ$, or from the point of view of types that its value has type Δ. The projections are:

$$fst : \Pi(\Delta, \varepsilon) \rightarrow \Delta \qquad snd : \Pi(\varepsilon, \Delta) \rightarrow \Delta$$

We check that *fst* can be applied to both X and Y but *snd* only to Y. In this view $\bullet$ and $\circ$ are the positions of a switch which tells whether the field is filled or empty. In fact it is better to distinguish between the value $\bullet$ which position the switch of a non empty field and a value V of type *void* which fills it. In fact V is the value returned by projections:

$$fst : \Pi(\Delta, \varepsilon) \rightarrow void \qquad snd : \Pi(\varepsilon', \Delta) \rightarrow void$$

Variables ε and ε' range over Δ and ∇ but not over *void*, they are of a different kind, say *flag*, while variables $\alpha, \beta, \ldots$ are of the kind *type* and range over *void* and the arrow and record types. We think of *void*, Δ or ∇ indifferently as sorts.

When values of fields are no more restricted to be only V, it is clear that we must put into fields both a switch and a value. We use an undefined value Ω which has all types (all sorts of the kind *type*) to fill previously empty fields. Records usually defined by

$$X = \boxed{1}\boxed{} \qquad Y = \boxed{2}\boxed{V}$$

now stand for

$$X = \boxed{\bullet, 1}\boxed{\circ, \Omega} \qquad Y = \boxed{\bullet, 2}\boxed{\bullet, V}$$

and have types

$$X : \Pi(\varepsilon.num, \nabla.\alpha) \qquad Y : \Pi(\varepsilon.num, \varepsilon'.void)$$

where "." is an infix product constructor of types. The first projection has type $\Pi(\Delta.\alpha, \varepsilon.\beta) \rightarrow \alpha$ and can be applied to both X and Y.

When labels are more numerous, but still finite, records can still be thought as a huge box with as many fields as the number of labels. In this view labels are just a syntactic way of specifying the significant fields. For instance, if labels are all strings with at most six letters, the notation $\{A = V\}$ stands for:

$$\{A = \bullet, V; B = \circ, \Omega; C = \circ, \Omega; \ldots zzzzzz = \circ, \Omega\}$$

and has type:

$$\Pi(\varepsilon.void, \nabla.\beta_1, \nabla.\beta_2, \ldots \nabla.\beta_{642544811})$$

Fortunately, the user will never see these structures but a more compact representation with labels, which reflects the way they are encoded. This encoding even allows to deal with a denumerable set of labels and will be described below.

Variants are labelled sums. The inclusion mechanism for variants can be studied in the restricted language. The switch of a sum field tells if the value must (then it is Δ), may (it is a variable), or must not (it is ∇) be injected into this field. For instance with two fields *left* and *right* we have the following types:

$$\begin{array}{lcl} in^{left} & : & void \rightarrow \sum(\Delta, \varepsilon) \\ in^{right} & : & void \rightarrow \sum(\varepsilon, \Delta) \\ out^{left} & : & \sum(\Delta, \nabla) \rightarrow void \end{array}$$

Variants are to concrete data types what records are to named labelled products. Concrete data types can be defined recursively, and variants encoding them will be recursive. So we extend ordinary types to recursive types. We formally present a language of expressions, a language of types which is kinded and regular, and show how Milner's type inference algorithm still prevails. Only the case of a finite ordered set of labels is studied, but an extension to a denumerable set of labels is suggested.

2 A formulation of records

2.1 Language of type expressions

We give a complete formulation of the intuitive approach of the previous section, embedding the simple types of ML into a more structured world of sorts. The language of sorts is ***kinded*** in order to control the range of variables, and it is regular so that it allows recursive types.

We define a language **K** of kinds as follows: the basic kinds are *type*, *flag* and *field* and the only kind constructor is $\Rightarrow$ of arity two. The language **K** is the closure of the set $\{type, field, flag\}$ by the arrow constructor:

$$\textit{if } u \in \mathbf{K} \textit{ and } v \in \mathbf{K} \textit{ then } (u \Rightarrow v) \in \mathbf{K}$$

We write

$$(u_1 \otimes u_2 \otimes \ldots u_n \Rightarrow v)$$

for

$$(u_1 \Rightarrow (u_2 \Rightarrow \ldots (u_n \Rightarrow v) \ldots))$$

Let L be a finite set of labels and l be its cardinality, $\mathcal{B}$ be a set of basic constants of the kind *type*, $\mathcal{C}$ be

$$\begin{array}{lll}
in^i & : & \alpha \to \sum(\varphi_1, \dots, \varphi_{i-1}, \Delta.\alpha, \varphi_{i+1} \dots \varphi_l) \\
out^i & : & \sum(\nabla.\beta_1, \dots, \nabla.\beta_{i-1}, \Delta.\alpha, \nabla.\beta_{i+1} \dots \nabla.\beta_l) \to \alpha \\
case^i & : & (\alpha \to \gamma) \to (\sum(\varphi_1, \dots \varphi_l) \to \gamma) \to \sum(\varphi_1, \dots, \varphi_{i-1}, \Delta.\alpha, \varphi_{i+1} \dots \varphi_l) \to \gamma \\
 & & \\
null & : & \prod(\nabla.\beta_1, \dots \nabla.\beta_l) \\
extract^i & : & \prod(\varphi_1, \dots, \varphi_{i-1}, \Delta.\alpha, \varphi_{i+1} \dots \varphi_l) \to \alpha \\
forget^i & : & \prod(\varphi_1, \dots \varphi_l) \to \prod(\varphi_1, \dots, \varphi_{i-1}, \nabla.\alpha, \varphi_{i+1} \dots \varphi_l) \\
new^i & : & \prod(\varphi_1, \dots \varphi_l) \to \alpha \to \prod(\varphi_1, \dots, \varphi_{i-1}, \varepsilon.\alpha, \varphi_{i+1} \dots \varphi_l)
\end{array}$$

Figure 2: Primitives for records and variants

the set of constructors $\{\to, \Pi, \Sigma, \Delta, \nabla\}$ which have the kinds:

$$\begin{array}{lll}
\to & : & type \otimes type \Rightarrow type \\
\prod & : & \underbrace{field \otimes field \otimes \dots field}_{l} \Rightarrow type \\
\sum & : & \underbrace{field \otimes field \otimes \dots field}_{l} \Rightarrow type \\
. & : & flag \otimes type \Rightarrow field \\
\Delta & : & flag \\
\nabla & : & flag
\end{array}$$

The constructors $\to$ and . are infixed. We denote by $\mathcal{S}$ the union $\mathcal{C} \cup \mathcal{B}$ of symbols. Let $\mathcal{V}^t$, $\mathcal{V}^f$ and $\mathcal{V}^b$ be three denumerable sets of sort variables of respective kinds *type*, *field* and *flag*. We note $\mathcal{V}$ their union.The set $\mathcal{R}$ of sorts is the set of first order kinded regular trees constructed over the set of variables $\mathcal{V}$ and the set of symbols $\mathcal{S}$. In order to extend expressions to generic expressions, we introduce three other denumerable sets of generic sort variables of the three basic kinds and name them by $_g$ subscripts. The set $\mathcal{R}_g$ is defined as the set $\mathcal{R}$, but replacing the sets of non generic variables by their unions with the associated sets of generic variables. Note that $\mathcal{V} \cap \mathcal{V}_g$ is empty but $\mathcal{R}$ is included in $\mathcal{R}_g$.

We denote sort variables by letters α, β, γ adding a subscript $_g$ for generic sort variables, and sorts by σ, τ adding a subscript if we admit generic sorts. We also use ϕ, ψ but only for variables of the kind *field*, and ε, δ for variables of the kind *flag*. We note $\langle\sigma_g\rangle$ the set of all sort variables occurring in σ_g. All variables used in toplevel definitions are obviously generic, but we will omit the $_g$ subscripts in figures.

2.2 Results on regular trees

We recall below some results on regular trees. Most of them are proved in [Huet].

- A graft is a mapping from $\mathcal{V}$ to $\mathcal{R}$ respecting the kinds. In the following we are only interested in finite grafts, i.e. the grafts μ such that the set $\{\alpha \mid \alpha\mu \neq \alpha\}$ is finite. Then μ is representable by the finite set of pairs: $\{(\alpha, \alpha\mu) \mid \alpha\mu \neq \alpha\}$.
- Rational trees can be seen as the application of a (finite) graft to a fixed variable ϵ, thus they are finitely representable by a variable ϵ and a set of pairs (α, σ).
- **Unification** There exists an algorithm U such that given two regular trees σ and τ:
  - if the two trees are not unifiable then U fails.
  - Otherwise U returns a most general unifier u, i.e. a graft such that if there exists a graft ν which unifies σ and τ then there exists another graft ξ such that $\nu = u\xi$.

  In fact algorithm U is just the usual algorithm where the occur test has been removed. We note $\sigma_g \overset{u}{\sim} \tau_g$ when the two generic sorts σ_g and τ_g are unifiable and u is their most general unifier.

We distinguish several classes of grafts:

- Grafts from $\mathcal{V}$ to $\mathcal{R}$ are denoted by μ, ν, ξ, ρ.
- Grafts from $\mathcal{V}_g$ to $\mathcal{R}_g$ have subscript.
- A principal unifier is noted u,v,....

Grafts are extended to automorphisms of $\mathcal{R}$ or $\mathcal{R}_g$. We note $\sigma\mu$ the application of the graft μ to the tree σ, and $\mu\nu$ for $\nu \circ \mu$ in such a way that $\sigma\mu\nu$ means $(\sigma\mu)\nu$ as well as $\sigma(\mu\nu)$.

$'\ell_i\ x$	$\equiv$	$In^i\ x$
let $'\ell_i\ y = x$ in y	$\equiv$	$Out^i\ x$
function $'\ell_i\ y \rightarrow f\ y$	$\equiv$	$f \circ Out^i$
function $'\ell_i\ y \rightarrow f\ y \mid x \rightarrow c\ x$	$\equiv$	$Case^i\ f\ c$
$\{\,\}$	$\equiv$	$null$
$\{r$ with $\ell_i = x\}$	$\equiv$	$new^i\ r\ x$
$\{\ell_{i_1} = x_1, \ldots\ \ell_{i_n} = x_n\}$	$\equiv$	$\{\{\ell_{i_1} = x_1\ \ldots\ \ell_{i_{n-1}} = x_{n-1}\}$ with $\ell_{i_n} = x_n\}$
$r.\ell_i$	$\equiv$	$extract^i\ r$
$\{r$ without $\ell_i\}$	$\equiv$	$forget^i\ r$

Figure 3: Syntax for records and variants

2.3 Syntax of expressions

We use a very simple language defined by the following grammar:

$$\begin{array}{rcll} e & := & b & \text{constant} \\ & \mid & x & \text{variable} \\ & \mid & \lambda x.e & \text{abstraction} \\ & \mid & e\ e & \text{application} \\ & \mid & \mu x.e & \text{recursion} \\ & \mid & \text{let } x = e \text{ in } e & \text{local definition} \end{array}$$

We have no special construction for products and variants but a finite collection of primitive functions (constants), which are listed in figure 2 with their respective sorts. Combining these primitives we can think of record and variant constructions as macro-syntax facilities, summarized in figure 3. One can change or add basic constructions just by modifying the set of primitives. For instance we could take:

$$\begin{array}{rl} new^i : & \prod(\varphi_1, \ldots, \varphi_{i-1}, \nabla.\beta, \varphi_{i+1} \ldots \varphi_l) \rightarrow \alpha \\ & \rightarrow \prod(\varphi_1, \ldots, \varphi_{i-1}, \varepsilon.\alpha, \varphi_{i+1} \ldots \varphi_l) \end{array}$$

which would restrict the $\{r$ with $\ell_i = x\}$ construction to be W^-. The other choice:

$$\begin{array}{rl} new^i : & \prod(\varphi_1, \ldots \varphi_l) \rightarrow \alpha \\ & \rightarrow \prod(\varphi_1, \ldots, \varphi_{i-1}, \Delta.\alpha, \varphi_{i+1} \ldots \varphi_l) \end{array}$$

would restrict the inclusion to semi-inclusion. Remark that non generic accesses of semi-inclusive fields can be compiled more efficiently.

2.4 Inference rules

We say that τ_g is a generic instance of σ_g if and only if there exists a graft μ_g such that $\tau_g = \sigma_g \mu_g$. Inference rules are applied to triples (A, e, τ_g) also called *judgements* and noted $A \vdash e : \tau_g$ where:

- A is an environment, i.e. a set of assertions $x : \sigma_g$ where the expression x is either a constant or a variable. We note A_x the set of all assertions in A which do not contain the expression x. We also note $\langle A \rangle$ the set of sort variables occurring in A.
- e is an expression
- τ_g is a generic sort expression

The system (Σ) of inference rules is:

$$(TAUT)\quad \frac{}{A \vdash x : \sigma_g} \quad (x : \sigma_g \in A)$$

$$(INST)\quad \frac{A \vdash e : \sigma_g}{A \vdash e : \sigma_g \mu_g}$$

$$(GEN)\quad \frac{A \vdash e : \sigma_g}{A \vdash e : \sigma_g[\alpha_g/\alpha]} \quad (\alpha \notin \langle A \rangle)$$

$$(FUN)\quad \frac{A_x \cup \{x : \sigma\} \vdash e : \tau}{A \vdash \lambda x.e : \sigma \rightarrow \tau}$$

$$(APP)\quad \frac{A \vdash e : \sigma \rightarrow \tau \quad A \vdash e' : \sigma}{A \vdash (e\ e') : \tau}$$

$$(LET)\quad \frac{A \vdash e : \sigma_g \quad A_x \cup \{x : \sigma_g\} \vdash e' : \tau}{A \vdash \text{let } x = e \text{ in } e' : \tau}$$

$$(MU)\quad \frac{A_x \cup \{x : \sigma\} \vdash e : \sigma}{A \vdash \mu x.e : \sigma}$$

We extend grafts to expressions with identity. Then the grafting μ of an assertion $x : \tau_g$ is the assertion $x : \tau_g \mu$ and the grafting of a judgement $A \vdash e : \sigma_g$ is the judgement $A\mu \vdash e : \sigma_g \mu$. The inference system is just Milner's system where types have been

$$(NEW^{W}) \quad \frac{x : \sigma_g \in A}{A \vdash^{W} x : \lfloor \sigma_g \rfloor}$$

$$(FUN^{W}) \quad \frac{(A_x \cup \{x : \alpha\})\nu \vdash^{W} e : \tau}{A\nu \vdash^{W} \lambda x.e : \alpha\nu \longrightarrow \tau}$$

$$(APP^{W}) \quad \frac{A\nu \vdash^{W} e : \tau \quad A\nu\nu' \vdash^{W} e' : \tau' \quad \tau\nu' \overset{u}{\sim} \tau' \longrightarrow \alpha}{A\nu\nu' u \vdash^{W} (e\ e') : \alpha u}$$

$$(LET^{W}) \quad \frac{A\nu \vdash^{W} e : \tau \quad (A_x \cup \{x : \lceil A\nu, \tau \rceil\})\,\nu' \vdash^{W} e' : \tau'}{A\nu\nu' \vdash^{W} \text{let } x = e \text{ in } e' : \tau'}$$

$$(MU^{W}) \quad \frac{(A_x \cup \{x : \alpha\})\nu \vdash^{W} e : \tau \quad \alpha\nu \overset{u}{\sim} \tau}{A\nu u \vdash^{W} \mu x.e : \tau u}$$

Figure 4: Rules defining the algorithm W.

replaced by kinded recursive kinds, and substitutions by kinded grafts. We note:

- $\lceil A, \sigma_g \rceil$ the generalization of σ_g in the context A, i.e. the grafting of all non generic variables of σ_g which do not appear in the context A by new generic variables.
- $\lfloor \sigma_g \rfloor$ the instantiation of σ_g, i.e. the grafting of all generic variables of σ_g by new non generic variables.

Both instantiation and generalization are defined modulo a renaming of variables. We mean by new variables some which occur neither in the context A nor in the sort σ_g. In fact we should note $\lfloor A, \sigma_g \rfloor$ rather than $\lfloor \sigma_g \rfloor$ since the instantiation also depends on the context A, but in a less important way than the generalization. If an instantiation or a generalization occurs more than once in the same phrase it will denote the same tree in all occurences.

Lemma 1 (SUB) *The system Σ is stable under non-generic grafts:*

$$A \vdash e : \sigma \Longrightarrow A\mu \vdash e : \sigma\mu$$

We define an algorithm W which for any environment A and expression e either fails or returns a pair (ν, τ). We note $A\nu \vdash^{W} e : \tau$ instead of $(\nu, \tau) = W(A, e)$. See figure 4.

Theorem 1 (Soundness of W) *If $A\mu \vdash^{W} e : \sigma$ then $A\mu \vdash e : \sigma$.*

Theorem 2 (Completeness of W) *For any judgement A and any expression e, if there exist a graft μ and a generic sort σ_g such that $A\mu \vdash e : \sigma_g$*

- *there exist ν and τ such that $A\nu \vdash^{W} e : \tau$*
- *there exists ξ such that $A\mu = A\nu\xi$ and $\lfloor \sigma_g \rfloor = \tau\xi$.*

Theorem 3 (Principal sort schemes) *For any environment A and expression e, either e is untypable under some graft of A and $W(A, e)$ fails or $W(A, e)$ succeeds with (μ, τ) and $\lceil A\mu, \tau \rceil$ is a principal sort scheme of e under $A\mu$.*

Proofs just consist in checking that the introduction of several kinds and the replacement of terms by trees do not alter the proofs with Milner's system. These use only the unification theorem and the substitution lemma. If we just replace "term" by "kinded regular tree" and "substitution" by "kinded graft", the lemma and the unification theorem are still valid, and so is Milner's proof.

Obviously, the proofs neither depend on the present kinds of sort constructors excepted the arrow one, nor of the sorts of the primitive values of the language of expressions. We can freely change the semantics of basic constructions or add new ones (the set of primitives we gave do not use all the possibilities of the sorts) by changing simultaneously the semantics and the sorts of their primitives in a consistent way. One can also keep kinded but non recursive sorts, or add some restriction on the construction of recursive sorts, provided that this invalids neither the substitution lemma nor the unification theorem.

3 Examples

We implemented a typechecking algorithm for our type system in CAML. The toplevel loop does not evaluate expressions but returns either an exception or a list of identifiers with their types. The identifier `it` is used when an expression is unnamed. The syntax of expressions includes the constructions for variants and records, and allows pattern matching in the usual way. A typable toplevel expression is always typed with a sort of kind *type* which is represented by a finite tree and a set of equations between variables and finite trees. The syntax of flags is + for Δ, - for ∇ and names of flag variables are only printed when they occurs more than once in the same phrase, so "." stands for an arbitrary *flag* variable. Most of the fields have the same scheme (each instanciated with distinct variables), and are gathered.

- the closed scheme $\nabla.\alpha$ is the default one and is not represented.
- the open scheme $\varepsilon.\alpha$ is represented by a row variable or by "..." if it occurs only once in a phrase. Otherwise two occurrences of the same scheme will assign the same flags and types to the same projections.

Schemes are just abbreviations for the regular field projections. Syntactically they correspond to the row expressions used by Wand [Wand87] but also by R. Stansifer [Sta88] and L. Jategaontar and J. Mitchell [JM88].

Record objects have always closed types:

```
#let r = {a=true; b=1};;
r :  {a: .bool, b: .num}
```

they can be extended with other fields very freely.

```
#{r with b=();c=3};;
it :  {a: .bool, b: .void, c: .num}
```

This allows the field being already present, even with an incompatible type. The significant fields have polymorphic flags and thus can be forgotten. The infix operator ~, defined by:

```
let x ~ y = if true then x else y;;
~ : 'a -> 'a -> 'a
```

is used to shorten examples.

```
#r ~ {b=1};;
it :  {a: -.bool, b: .num}
```

Notice that the typechecker keeps trace of the types that have been put into fields. This avoids the strange property of Amber [Car86] which assigns a type to:

```
({a=true} ~ {a=true; b=1}) ~ {a=true; b=()}
```

but fails with:

```
{a=true} ~ ({a=true; b=1} ~ {a=true; b=()})
```

which is rather surprising. It also seems to prevent us from mixing labels which should have nothing in common, but this is rather a restriction of inclusion. We shall come back to it later. The *without* construction forgets a field explicitly in which case it is regenerated, i.e. its type becomes a fresh variable.

```
#{r without b};;
it :  {a: .bool, b: -.'a}
```

First, the field φ of the b projection is withdrawn from the scheme if it was not there yet, then the scheme is used to create the type of the new record, so that further constraints on it will not affect φ:

```
#fun r r' ->
# {r without option} ~ {r' without option};;
it :
 {option: .'a , p...} ->
 {option: .'b , p...} -> {option: -.'c , p...}
```

Fields are recalled either by pattern matching or by one of the projections:

```
#fun r -> r.a;;
it :  {a: +.'a , ...} -> 'a
```

Another restriction is due to non genericity of λ-bound values. We fail with:

```
#fun x -> if x.a then x else {b=1};;
Typing failed
```

although we succeed with:

```
#let x = {a=true;b=2}
#in if x.a then x else {b=1};;
it :  {a: -.bool, b: .num}
```

This restriction is inherent to ML polymorphism which stops at the lambda boundaries. The same situation arises in:

```
#fun f -> if f 1 then f 1 else f();;
```

which is not typable though:

```
#let f _ = true in if f 1 then f 1 else f();;
```

is, indeed. The above counter-example could be solved by rewriting the lambda expression into a let expression, but this is not the general case.

Variant objects have naturally open types (they can be constrained to be closed). They are created with constructors which are quoted labels.

```
#let s = 'A 1;;
s :  [A: +.num | ...]
```

Sums are destructured only by pattern matching:

```
#function 'A x -> x | 'B() -> true;;
it :  [A: .bool | B: .void] -> bool
```

A "catch all" indicated by "_" matches all variants.

```
#let trap_all = function 'B x -> x | _ -> 0;;
trap_all :  [B: .num | ...] -> num
```

Recursive functions usually have recursive variant types:

```
#let rec map foo = function
#  'Nil()   -> 'Nil()
#| 'Cons l  ->
#   'Cons {hd= foo l.hd; tl= map foo l.tl};;
map :
 ('a -> 'b) -> 'c -> 'd
 with
  'd =
    [Cons: +.{hd: .'b, tl: .'d} | Nil: +.void |
     ...]
  'c =
    [Cons: .{hd: +.'a, tl: +.'c , ...} |
     Nil: .void]
```

Types are powerful enough to tell that a list must be non empty. The function:

```
#let hd ('Cons r) = r.hd;;
hd :  [Cons: .{hd: +.'a , ...}] -> 'a
```

cannot be applied to the null list, but the function Hd defined by:

```
#let check_cons = function
#  'Cons l -> 'Cons l
#| 'Nil () -> failwith "check_cons";;
check_cons :
 [Cons: .'a | Nil: .void] -> [Cons: +.'a | ...]

#let Hd = hd o check_cons;;
Hd :
  [Cons: .{hd: +.'a , ...} | Nil: .void] -> 'a
```

will accept a null list and raise an exception. Dynamic typechecking during pattern matching can be dissociated from the active part of the function. Longer examples are given in the appendix.

We previously mentioned the possibility of changing the semantics of the basic constructions, or adding new ones. In previous examples the flag of a field could never depend on the flag of another field. The reason is that all primitives already have this property. But there are interesting constructions where this is no longer true, for instance a primitive which exchanges two fields of a record.

```
#fun r -> {r where gnu and gnat permuted};;
it :
 {gnu: u.'a, gnat: v.'b, ...p} ->
 {gnu: v.'b, gnat: u.'a, ...p};;
```

This operation cannot be typed in systems which encode inclusion of records with a global inclusion relation. With variants, similar constructions have also great interest. The function

```
#function 'True()  -> 'False()
#       | 'False() -> 'True();;
it :
 [False: .void | True: .void] ->
 [False: +.void | True: +.void | ...]
```

could be written as

```
#transposition 'True()  -> 'False()
#             | 'False() -> 'True();;
it :
 [False: u.void | True: v.void] ->
 [False: v.void | True: u.void | ...]
```

and so better typed. We let the reader build his own examples.

4 Recovering the full power of inclusion

The system proposed is limited to a finite set of labels. This is sufficient from a pragmatic point of view since in any real language, only a finite number of operations will ever be computed. It is always possible to reason *a posteriori* in a large enough set to assert that what has been done is correct. From a theoretical point of view an infinite set of labels is obviously better and avoids the above meta-reasoning. Note that in ML with concrete type constructors, a type declaration is usually but not essentially a meta operation.

$$\begin{aligned}
&\textit{null} &&: \prod(\nabla.\beta_1, \dots \nabla.\beta_l)\\
&\textit{extract}^i &&: \prod(\varphi_1, \dots, \varphi_{i-1}, \Delta.\alpha, \varphi_{i+1} \dots \varphi_l) \to \alpha\\
&\textit{forget}^i &&: \prod(\varphi_1, \dots \varphi_l) \to \prod(\varphi_1, \dots, \varphi_{i-1}, \nabla.\alpha, \varphi_{i+1} \dots \varphi_l)\\
&\textit{new}^i &&: \prod(\varphi_1, \dots \varphi_l) \to \alpha \to \prod(\varphi_1, \dots, \varphi_{i-1}, \Diamond.\alpha, \varphi_{i+1} \dots \varphi_l)
\end{aligned}$$

Figure 5: Primitives with structural subtyping

$$\begin{aligned}
&\textit{null} &&: \prod(\nabla, \dots \nabla)\\
&\textit{extract}^i &&: \prod(\varphi_1, \dots, \varphi_{i-1}, \Delta\alpha, \varphi_{i+1} \dots \varphi_l) \to \alpha\\
&\textit{forget}^i &&: \prod(\varphi_1, \dots \varphi_l) \to \prod(\varphi_1, \dots, \varphi_{i-1}, \nabla, \varphi_{i+1} \dots \varphi_l)\\
&\textit{new}^i &&: \prod(\varphi_1, \dots \varphi_l) \to \alpha \to \prod(\varphi_1, \dots, \varphi_{i-1}, \Diamond\alpha, \varphi_{i+1} \dots \varphi_l)
\end{aligned}$$

Figure 6: Primitives with stronger subtyping

Kinded regular trees could be extended to finitely generated kinded regular trees. We could prove a unification theorem for them similar to the one for regular trees and the substitution lemma would not be altered, so the results would still hold. Moreover, this proof would justify the schemes we used in the implementation to abbreviate the non significant fields into a compact representation.

The generality of our system is preserved for labelling. But there are two limits in the encoding of inclusion.

- ML polymorphism is too restrictive to code the full power of inclusion.
- Polymorphism can only code structural inclusion.

However the approach followed here has interesting points:

- From both a theoretical and practical point of view, it is very simple.
- It is also modular because the mechanism of inclusion on records is not an ***ad hoc*** assertion but is built in.
- It allows interesting constructions which are not typable in a system where inclusion on records is defined globally, because there is no way to specify any relation between fields.

So we try to recover the full power of inclusion keeping these properties.

4.1 Records with structural subtyping

The first limitation can be re-examined in the simple language used in the intuitive approach. The important idea was to code the records:

$$X = \boxed{\bullet}\boxed{} \qquad\qquad Y = \boxed{\bullet}\boxed{\bullet}$$

as the total functions:

$$X = \boxed{\bullet}\boxed{\circ} \qquad\qquad Y = \boxed{\bullet}\boxed{\bullet}$$

We first proposed to type X and Y with:

$$X : \Pi(\Delta, \nabla) \qquad\qquad Y : \Pi(\Delta, \Delta)$$

but X could not be used instead of Y.

Then we explained how to encode inclusion with polymorphism. If the simple language has inclusion between atomic types, we introduce a new basic flag $\Diamond$ (read **up and down**) as a subflag of both Δ and ∇. Then everything works as before. More precisely in a language (IS) primitive functions would have the sorts given in figure 5. In fact identifying Δ and $\Diamond$ leads to a very similar though weaker system. It still codes $(IR_1^\infty W)$ inclusion but cannot encode semi-inclusion any longer.

4.2 Records with stronger subtyping

We cannot yet forget the types of fields, but only their flags. When we introduce multiple values we decided to put into fields both switches and values. Another solution consists in putting values only in fields which have an open switch. From this point of view the kind of fields is no longer a pair of a flag and a type, but either $\Delta\alpha$ or ∇. Obviously we add a third case $\Diamond\alpha$ which is a subfield of both previous ones, we have:

$$\Diamond\alpha \subseteq \Delta\alpha \qquad \Diamond\alpha \subseteq \nabla$$

We code the records:

$$X = \begin{array}{|c|c|}\hline 1 & \\ \hline\end{array} \qquad Y = \begin{array}{|c|c|}\hline 2 & V \\ \hline\end{array}$$

as the total functions:

$$X = \begin{array}{|c|c|}\hline \bullet 1 & \circ \\ \hline\end{array} \qquad Y = \begin{array}{|c|c|}\hline \bullet 2 & \bullet V \\ \hline\end{array}$$

which have types:

$$X : \Pi(\Diamond num, \nabla) \qquad Y : \Pi(\Diamond num, \Diamond void)$$

More precisely, primitive functions would have the sorts of figure 6.

We shall identify Δ and $\Diamond$. The inclusion is no longer atomic and thus cannot be treated by structural subtyping. However it is a subcase of general inclusion studied by Y-C. Fuh and P. Mishra. Inference of a set of constraints can be solved as in [FM88]. Checking the consistency can be done using the matching of [FM88] and looking for a solution with lower fields. But we shall have to find efficient algorithms to simplify the set of constraints.

The difficulty is shown by the example:

```
#fun x -> {a=x; b=x} ~ {a=1; b=()};;
```

The type of this function is:

$$\alpha \rightarrow \Pi(\varphi, \psi) \; with \begin{cases} \Delta num \subseteq \varphi \\ \Delta\alpha \quad\;\; \subseteq \varphi \\ \Delta void \subseteq \psi \\ \Delta\alpha \quad\;\; \subseteq \psi \end{cases}$$

The set of constraints is not simplifiable, since both variables φ and ψ can be "up". Remark that this checking could be expensive in time. Moreover, only one field can be "up" at a time, but this piece of information cannot be coded with containment. Thus type containment may not be a good way to express the power of the language (IS^+). A good candidate might be a restriction of Coppo's system [Coppo], but this needs investigating.

Conclusion

We presented a new solution to typecheck records and variants, which seems to have interesting properties: it captures the essential notions of inclusion, but in a natural extension of ML polymorphism, and infers more precise and recursive types which makes concrete type declarations optional. Most of these ingredients are very modular, and thus the language designer may choose his own version.

We also checked that the algorithm allows the typing of huge expressions in reasonable time and is even very competitive with the current algorithm used in CAML. Type expressions are much more complex, but much of the information is not essential to the user. It seems reasonable to show him only partial types, unless he explicitly asks for complete information.

We limited this discussion to a very simple version of polymorphism where ML type inference algorithm is still applicable. The advantage is both a theoretical and practical simplicity, but one of the drawbacks is a limitation of inclusion. The method proposed is more general, and can be applied to systems with subtypes where the power of inclusion is increased.

References

[CAMLr] Pierre Weis. "The CAML Reference Manual". INRIA 1987.

[CAMLp] Guy Cousineau and Gérard Huet. "The CAML Primer". INRIA 1987.

[Car84] Luca Cardelli. "A Semantics of Multiple Inheritance". In Information and Computation 1988. In Semantics of Data Types, Lecture Notes in Computer Science n. 173, Springer Verlag, 1983.

[Car86] Luca Cardelli. "Amber". In Combinators and Functional Programming Languages, Proceedings of the 13th Summer School of the LITP, Le Val D'Ajol, Vosges, France, May 1985, Lecture Notes in Computer Science n. 242, Spinger Verlag, 1986.

[Car88] Luca Cardelli. "Structural Subtyping and the notion of Power Type". In Proceedings of the Fifteenth Annual Symposium on Principles Of Programming Languages, 1988.

[Coppo] Mario Coppo. "An Extended Polymorphic Type System for Applicative Languages". In Proceedings of MFCS '80, Lectures Notes in Computer Science n. 88, Springer Verlag, pages 194-204.

[CW85] Luca Cardelli and Peter Wegner. "On understanding types, data abstraction, and polymorphism". Computing Surveys, vol. 17(4). 1985.

[FM88] Y-C. Fuh and P. Mishra. "Type inference with subtypes". In Proceedings of ESOP '88, Lecture Notes in Computer Science n. 300, Springer Verlag, pages 94-114, 1988.

[Huet] Gérard Huet. "Résolution d'équations dans les langages d'ordre $1, 2, \ldots, \omega$". Thèse de doctorat d'état, Université Paris 7, 1976.

[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.

[KTU88] A.J. Kfoury, J. Tiuryn and P. Urzyczyn. "On The Computational Power of Universally Polymorphic Recursion". In Proceedings of the Third Symposium on Logic In Computer Science, 1988.

[Mit84] John C. Mitchell. "Coercion and Type Inference". In Proceedings of the Eleventh Annual Symposium on Principles Of Programming Languages, 1984.

[Mit88] John C. Mitchell. "Polymorphic Type Inference". In Information and Computation, 1988.

[Sta88] Ryan Stansifer. "Type inference with Subtypes". In Proceedings of the Fifteenth Annual Symposium on Principles of Programming Languages, San Diego, California, 1988.

[Wand87] Mitchell Wand. "Complete type inference for simple objects". In Proceedings of the Second Symposium on Logic In Computer Science, 1987.

[Wand88] Mitchell Wand. "Corrigendum: Complete type inference for simple objects". In Proceedings of the Third Symposium on Logic In Computer Science, 1988.

Appendix

The following program implements the quicksort algorithm. In this example booleans are considered as sums. The additional syntax is that `x::y` stands for `'Cons{hd=x, tl=y}`, in both patterns and expressions.

```
#let select p =
# let rec select_rec =
# function
#   'Nil()  -> failwith "select"
# | 'Cons l ->
#    if p l.hd
#     then l
#    else (let L = select_rec l.tl in {hd= L.hd; tl= l.hd::L.tl})
# in select_rec
#;;
select :
 ('a -> bool) -> 'b -> 'c
 with 'b = [Cons: +.'c | Nil: .void]
 and 'c = {hd: +.'a, tl: +.'b}

#let partition  p l =
# list_it
#  (fun a l ->
#    if p a then {hd= a::l.hd; tl= l.tl} else {hd= l.hd; tl= a::l.tl})
#  l {hd= 'Nil; tl= 'Nil}
#;;
partition :
 ('a -> bool) -> 'b -> {hd: +.'c, tl: +.'d}
 with
  'c = [Cons: +.{hd: .'a, tl: .'c} | Nil: +.void | ...]
  'd = [Cons: +.{hd: .'a, tl: .'d} | Nil: +.void | ...]
  'b = [Cons: .{hd: +.'a, tl: +.'b , ...} | Nil: .void]

#let sort_append  p le =
# let rec sort  =
# function
#   []         -> I
# | _::_  as x ->
#    let l = select p x in
#    let L = partition (fun y -> le y l.hd) l.tl in
#    (sort L.hd) o (cons l.hd) o (sort L.tl)
# in sort
#;;
sort_append :
 ('a -> bool) -> ('a -> 'a -> bool) -> 'b -> 'c -> 'c
 with
  'c = [Cons: +.{hd: .'a, tl: .'c} | ...]
  'b = [Cons: +.{hd: +.'a, tl: +.'b} | Nil: +.void]

#let sort  order list =
# sort_append (fun x -> true) order list 'Nil
#;;
sort :
 ('a -> 'a -> bool) -> 'b -> 'c
 with
  'c = [Cons: +.{hd: .'a, tl: .'c} | Nil: +.void | ...]
  'b = [Cons: +.{hd: +.'a, tl: +.'b} | Nil: +.void]
```