Objective Label 2.04: User's Manual

Jacques Garrigue

November 29, 1999

Contents

1 Introduction

O'Labl, for Objective Label, extends Objective Caml to labeled and optional parameters [2], and also adds polymorphic variants [3]. The modifications to the language itself are minimal, but using labels will deeply change the aspect of programs. They are more readable, easier to maintain, and one can really understand the meaning of a function by looking at its type.

Particularly interesting is the LablTk library, based on François Rouaix's CamlTk interface [5], which allows one to write ML programs in a style very close to Tcl/Tk's.

LablBrowser, based on it, demonstrates the power of typeful programming in the presence of labels: the integrality of O'Labl libraries is browsable and searchable, as well as user's programs, on the basis of typing information only.

The O'Labl distribution contains a patch to modify O'Caml[1], the library LablTk, and the module browser LablBrowser. See appendix A for installation.

In the following we suppose that you are at least familiar with a language of the ML family, preferably of the CAML branch.

2 Labeled and optional arguments

One of the goals of O'Labl is to show that labels and variants are not only useful, but also inexpensive: both in terms of the work necessary to extend the original compiler, and efficiency of the resulting code.

As a result, modifications to O'Caml were limited to a minimum, making this extension immediately usable by an O'Caml programmer. This section and the next one describe syntactic and semantic modifications to the language.

2.1 Labels, optionals and currying

Until now, no widely distributed functional language provided for labeled or optional arguments. This comes essentially from the fact these languages allow currying, and, as a result, partial application.

For instance, suppose we define[2],

```
let rec map fun:f = function
    [] -> []
  | x :: l -> f x :: map fun:f l
```

We can use it fully applied as `map fun:(fun x → x+1) [1;2;3]`, but also partially applied `map fun:(fun x → x+1)`, as usual.

But what to do with the unlabeled partial application `map [1;2;3]`? Intuitively, we expect it to have type **fun**`:(int → 'a) → 'a`, otherwise there is no point in having labels. However, compiling this without changing semantics is not trivial.

When we add optional parameters, the relation with currying is even more complex. Take the function `sort : ?`**order**`:('a → 'a → bool) → 'a list → 'a list`. The "?" in front of "**order**:" means that this argument is optional. We can do it, since O'Caml has polymorphic comparison: we take `(<) : 'a → 'a → bool` as default. Still, we have a problem with determining when we should decide this argument was omitted. Intuitively, `sort [1;4;3]` should directly give `[1;3;4]` as result. But, how should we handle `sort [1;4;3]` **order**`:(>)`? The trivial extension results in a type error, where one would want arguments to commute. We need a more precise definition of omission semantics.

[1] Currently version 2.04. Check that the O'Caml version and the patch version match.
[2] "**name**:" denotes a label of name **name**.

2.2 Syntax

The original syntax of O'Caml is extended with the following production rules (*cf.* O'Caml's reference manual for the syntax of the rules).

$$
\begin{array}{rcl}
\textit{typexpr} & ::= & \ldots \\
 & \mid & [?]\textit{typexpr} \to \textit{typexpr} \\
 & \mid & [?]\textbf{label}{:}\textit{typexpr} \to \textit{typexpr} \\
\textit{expr} & ::= & \ldots \\
 & \mid & \textit{expr}\ \textit{labeled-expr}^{+} \\
 & \mid & \texttt{fun}\ \{\textit{labeled-simple-pattern}\}^{+}\ [\texttt{when}\ \textit{expr}] \to \textit{expr} \\
 & \mid & \texttt{function}\ \textit{labeled-pattern}\ [\texttt{when}\ \textit{expr}] \to \textit{expr} \\
 & & \qquad \{\mid \textit{labeled-pattern}\ [\texttt{when}\ \textit{expr}] \to \textit{expr}\}^{*} \\
\textit{labeled-expr} & ::= & [?]\textit{expr} \\
 & \mid & [?]\textbf{label}{:}\textit{expr} \\
\textit{labeled-pattern} & ::= & \textit{pattern} \\
 & \mid & \textbf{label}{:}\textit{pattern} \\
 & \mid & ?\textit{pattern}\ [\texttt{[<}\ \textit{expr}\ \texttt{>]}] \\
 & \mid & ?\textbf{label}{:}\textit{pattern}\ [\texttt{[<}\ \textit{expr}\ \texttt{>]}]
\end{array}
$$

"**label**:" is a new lexical entity, "`[a-z][a-zA-Z0-9_']`***:**" being its defining regular expression. As such, there is no restriction as which identifiers you may use as labels: keywords of the language are welcome, like "**fun**:" above.

Since heavy use of labels may make a program more verbose than necessary, some abbreviations are also permitted. They are all based on the expansion rule

:name ⟶ **name**:name

and its optional version

?:name ⟶ ?**name**:name

This rule being applied at the lexical level, it is valid in all contexts: patterns, expressions and types.

There are some interferences between labels and the original syntax of O'Caml at three levels:

- when adding type constraints in a pattern (*pattern* : *typexpr*) or an expression (*expr* : *typexpr*), if ":" touches either the end of the pattern/expression or the beginning of the type, it will be parsed as part of a label and result in a parse error. Add spaces.
- when writing interfaces or record types, the sequence "value name (or field name) :" may be lexed as a label. Since the parser was extended to accept this case also, you need not worry about that.
- "`?`" was also used by stream parsers to denote error handlers. To avoid the ambiguity you must now use "`???`" for that.

2.3 Semantics

Note first that application is no longer a binary operation — the original definition is *expr expr* — but application of an expression to a list of labeled expressions[3]. This is to avoid the interference between order and optionals: *between arguments applied simultaneously on different labels, order is irrelevant.*

[3] Notwithstanding its formal definition, O'Caml was already using this definition of application internally.

Labeled arguments

In absence of optionals (ignore all the "?"), semantics is easy. You can apply a function on any of its labels, and in any order. The labeled argument will be substituted for the first parameter with same label. That is:

```
# let f = map [1;2;3];;
val f : fun:(int -> 'a) -> 'a list
# f fun:(fun x -> x+1);;
- : int list = [2; 3; 4]
```

When you have several arguments on the same label (quite rare, except for the empty label), they are just applied in the order given.

```
# let f a:x a:y = x - y;;
val f : a:int -> a:int -> int
# f a:3 a:1;;
- : int = 2
```

Optional parameters

One can introduce default values in a function definition.

```
# let f x ?incr:y [< 1 >] = x + y;;
val f : int -> ?incr:int -> int
# f 1;;
- : int = 2
```

As explained above, we need some definite rule to decide when an argument was omitted. This is rule is in some way arbitrary, but it appears to do what one wants to do in most cases.

1. When a function is passed as argument to another function, all its optional parameters are considered omitted.

```
# let g = (fun x -> x) f;;
val g : int -> int
# g 3;;
- : int = 4
```

2. For application, optional labels match their non-optional counterpart.

```
# f 1 incr:3;;
- : int = 4
# f incr:3;;
- : int -> int
```

3. After an application, all optional labels appearing at the head of the result type are considered omitted.

```
# f 1;;
- : int = 2
# let g () ?:x [< 1 >] ?:y [< 1 >] () ?:z [< 1 >] = x * y * z;;
val g : unit -> ?x:int -> ?y:int -> unit -> ?z:int -> int
# g ();;
- : unit -> ?z:int -> int
# g () () x:3;;
- : int = 3
```

Optionals as syntactic sugar

In fact default arguments are only syntactic sugar for simpler optional parameters. They are implemented as option types (`type 'a option = None | Some of 'a`).

```
# let f () ?:x = x;;
val f : unit -> ?x:'a -> 'a option
# f ();;
- : 'a option = None
# f () x:3;;
- : int option = Some 3
```

In order to allow some forms of meta-programming with optionals, "?" may also be used in applications, with two effects:

- The corresponding argument is passed as a "raw" parameter (without doing the above conversion). Also, if this argument is a function containing optionals, those are not discarded.

```
# f () ?x:None;;
- : 'a option = None
# fun ?:x -> f () ?:x;;
- : ?x:'a -> 'a option
```

- When put in front of *all* arguments, other optional parameters are not discarded.

```
# f ?();;
- : ?x:'a -> 'a option
```

About call-by-value

ML is defined as a language with call-by-value semantics. This defines an order of evaluation for expressions[4], permitting the integration of side-effects in the language. However this definition is made with respect to classical lambda-calculus. The same definition with label-selective lambda-calculus (as defined in [1]) would be very hard to implement efficiently — there should be different evaluation paths depending on the order the arguments were applied. As a result, what we get in O'Labl is call-by-value operating on the *translation into classical lambda-calculus* of the original program.

Usually, you will not see it: as soon as a closure gets all its necessary arguments, it is evaluated.

```
# let f () =
    print_endline "Hello!";
    fun :x :y -> x + y;;
val f : unit -> x:int -> y:int -> int
# f () y:1;;
Hello!
- : x:int -> int
# let g = f y:1 in g ();;
Hello!
- : x:int -> int
```

Both when "()" and "**y**:1" are applied simultaneously or separately, "Hello!" is printed as soon as "()" is passed. (However, the first form is more efficiently compiled.)

Things get a little strange when you use tracing.

[4] In O'Caml, this does not specify the order of evaluation of the arguments, nor whether the arguments should be evaluated before or after the function.

```
# trace f;;
f is now traced.
# f () x:1;;
f ⟵ ()
Hello!
f ⟶ <fun>
f* ⟵ x: 1
f* ⟶ <fun>
- : y:int -> int = <fun>
# f () y:1;;
f ⟵ ()
Hello!
f ⟶ <fun>
- : x:int -> int = <fun>
```

The function is correctly traced only when there is no "gap" in the parameters passed. In the second case, since **x**: precedes **y**:, the trace of **y**:1 is delayed until the resulting function is applied on **x**:. Since tracing is done by instrumenting the original function, and out-of-order application with a gap creates a closure around it, there is no easy way to get more accurate trace information.

If you are curious about what is exactly done by compilation, use `olabl` with option `-dlambda` : an intermediate form will be displayed.

2.4 Typing

Type checking is extended to allow these new constructs. We do not detail the modifications here, the formal definition can be found in [2].

Only one modification had to be made in an unrelated part of the type checker. The definition of value, used to restrict polymorphism to side-effect free expressions, was extended in the following manner:

- let $f, e_1, \ldots, e_n$ be values. If f is of type $\mathbf{p}{:}\theta \to \tau$, and $\mathbf{p} \notin \{\mathbf{q_1}, \ldots \mathbf{q_n}\}$, then $f\ \mathbf{q_1}{:}e_1\ \ldots\ \mathbf{q_n}{:}e_n$ is still a value.

This allows one to do some tricky things with optional parameters.

Another modification was done, in the typing of statement sequences, to avoid some stupid errors. When the type of the result of the expression preceding a semicolon is unknown, it is resolved as unit. When it is known, the old behaviour is kept, and you only get a warning when this is a partially applied function.

```
# fun x -> x; ();;
- : unit -> unit = <fun>
# fun (x : int) -> x; ();;
- : int -> unit = <fun>
# fun (x : int -> int) -> x; ();;
Warning: this function application is partial,
maybe some arguments are missing.
- : (int -> int) -> unit = <fun>
```

2.5 Interaction with objects

There are limitations on the use of labels and optionals when using them in combination with objects.

- Label reordering cannnot be done properly on methods. As a result, calling a method whose type is still unknown with a parameter order different from the original one may cause type mismatches. This does not happen if the type of the method is already known.

- If a method has optional parameters, the typechecker should know its type (or the class of the object) when you use it. This also applies to polymorphic methods (*cf.* section 4).
- When you define an object, all its methods are already known, so that you can call them freely through the self abbreviation.

```
# class point :x :y = object
    val mutable x = x
    val mutable y = y
    method move x:dx y:dy = x <- x + dx; y <- y + dy
    method omove  ?x:dx [< 0 >] ?y:dy [< 0 >] =
        x <- x + dx; y <- y + dy
  end;;
class point :
  x:int ->
  y:int ->
  object
    val mutable x : int
    val mutable y : int
    method move : x:int -> y:int -> unit
    method omove : ?x:int -> ?y:int -> unit
  end
# let move_x10 p = p#move y:0 x:10;;
Warning: this expression is not known to have a method move.
val move_x10 : < move : y:int -> x:int -> 'a; .. > -> 'a = <fun>
# move_x10 (new point x:5 y:3);;
This expression has type
  point = < move : x:int -> y:int -> unit;
            omove : ?x:int -> ?y:int -> unit >
but is here used with type
  < move : y:int -> x:int -> 'a; omove : ?x:int -> ?y:int -> unit >
# let omove_x10 (p : #point) = p#omove x:10;;
omove_x10 : #point -> unit = <fun>
```

move_x10 demonstrates the problem appearing with out-of-order application on unknown methods.

In omove_x10 we explicitly constrain p to be a subtype of point. Otherwise we would not know which arguments are optional and could not compile it properly. This formulation is still general since any subtype of point may be used with this function.

Notice the warning in the move_x10 example: as a policy, in O'Labl you should explicitly type all object arguments in functions. This is not imperatively needed for normal order calls, though not giving a type may delay type errors to unrelated parts of the program, and this is required for out-of-order calls, optionals, and polymorphic methods.

2.6 Suggestions for labeling

Like for names, choosing labels for functions is not an easy task. A good labeling is a labeling which

- makes programs more readable,
- is easy to remember,
- when possible, allows useful partial applications.

We explain here the rules we applied when labeling the standard library.

To speak in an "object-oriented" way, one can consider that each function has a main argument, its *object*, and other arguments related with its action, the *parameters*. To permit the combination of functions through functionals, the object will not be labeled. Its role is clear by the function itself. The parameters are labeled with keywords reminding either of their nature or role. The role is to prefer, since the nature will often be given by the type itself. Obscure abbreviations should be avoided.

```
List.map : fun:('a -> 'b) -> 'a list -> 'b list
output : out_channel -> buffer:string -> pos:int -> len:int -> unit
```

When there are several objects of same nature and role, they are all left unlabeled.

```
List.iter2 : fun:('a -> 'b -> 'c) -> 'a list -> 'b list -> unit
```

Choosing the object contains some part of arbitrary, readability and usability being the real goals. In `assoc` the theoretical object would probably be the association list, but the following choice seems more natural. This is also coherent with `mem`.

```
List.assoc : 'a -> in:('a * 'b) list -> 'b
List.mem : 'a -> in:'a list -> bool
```

When there is no object, all arguments are labeled,

```
open_out : name:string -> out_channel
Sys.rename : old:string -> new:string -> unit
```

however, when there is only one argument, it is often left unlabeled.

```
Format.open_hvbox : int -> unit
```

All these are only suggestions, but one shall keep in mind that the choice of labels is essential for readability. Omissions or bizarre choices will make the program difficult to maintain.

In the ideal, the right function name with right labels shall be enough to understand the function's meaning. Since one can get this information with LablBrowser or olabl, the documentation is only used when a more detailed specification is needed.

3 Polymorphic variants

Here we introduce the syntax of polymorhic variants, informally discuss their use, and explain how they are typed.

3.1 Syntax

The already extended syntax is again extended to include polymorphic variants.

$$
\begin{array}{rcl}
\textit{typexpr} & ::= & \ldots \\
 & | & [\,'\,a]\ \texttt{[}\textit{variant-kind}\ \texttt{]} \\
 & | & [(\,'\,a,\ldots)]\ \#\textit{ident}\ [\texttt{as}\ \ '\,a] \\
 & | & [(\,'\,a,\ldots)]\ \#\textit{ident}\ \texttt{[}>\textit{ident}^{+}\ \texttt{]}\ [\texttt{as}\ \ '\,a] \\
 & | & [(\,'\,a,\ldots)]\ \#\textit{ident}\ \texttt{[}\textit{ident}^{+}\ \texttt{]}\ [\texttt{as}\ \ '\,a] \\
\textit{variant-kind} & ::= & \textit{tag-type}^{*} \\
 & | & \textit{tag-type}^{*}\ ..\ [>\textit{ident}^{+}] \\
 & | & <\textit{tag-type}^{+}\ [>\textit{ident}^{+}] \\
 & | & >\textit{tag-type}^{+} \\
\textit{tag-type} & ::= & \textit{ident} \\
 & | & \textit{ident}\ \texttt{(}\textit{typexpr}\texttt{)} \\
\textit{expr} & ::= & \ldots\ |\ \grave{}\,\textit{ident simple-expr} \\
\textit{simple-expr} & ::= & \ldots\ |\ \grave{}\,\textit{ident} \\
\textit{pattern} & ::= & \ldots\ |\ \grave{}\,\textit{ident simple-pattern} \\
\textit{simple-pattern} & ::= & \ldots\ |\ \grave{}\,\textit{ident}
\end{array}
$$

A variant kind (eventually preceded by a type variable it constrains) is written between square brackets ("`[`" and "`]`").

Variant expression and pattern are identifiers (either low- or upper- case) preceded by a backquote ("`` ` ``") and eventually followed by an argument.

A variant kind can be seen as a triple (T,U,L). T is a set of bindings from tags to types (or $\emptyset$ if the tag takes no argument), U the upper bound of the variant (a set of tags, $\top$ if there is no bound), and L its lower bound. Here are examples of the different forms and their meanings.

	Syntax	Meaning
(a)	`[a b(int)]`	$(\{\mathtt{a} \mapsto \emptyset, \mathtt{b} \mapsto \mathtt{int}\}, \{\mathtt{a},\mathtt{b}\}, \{\mathtt{a},\mathtt{b}\})$
(b)	`[a b(int) .. > b]`	$(\{\mathtt{a} \mapsto \emptyset, \mathtt{b} \mapsto \mathtt{int}\}, \top, \{\mathtt{b}\})$
(c)	`[< a b(int) > b]`	$(\{\mathtt{a} \mapsto \emptyset, \mathtt{b} \mapsto \mathtt{int}\}, \{\mathtt{a},\mathtt{b}\}, \{\mathtt{b}\})$
(d)	`[> a b(int)]`	$(\{\mathtt{a} \mapsto \emptyset, \mathtt{b} \mapsto \mathtt{int}\}, \top, \{\mathtt{a},\mathtt{b}\})$

There is a double subtyping relation between variant kinds, such that $(a) < (c) < (b)$ and $(a) < (d) < (b)$. Kinds of the form of (a) are *static*: they are minimal in the subtype hierarchy, and cannot change anymore through type unification. `[..]` is the other extremity of the hierarchy, the kind we know nothing about.

In a way similar to object types, "#" introduces abbreviations for subtypes.

	Abbreviation	Extended form
(a)	`ab`	`[A(int) B]`
(b)	`#ab`	`[< A(int) B]`
(c)	`#ab[> A]`	`[< A(int) B > A]`
(d)	`#ab[A]`	`[A(int)]`

These abbreviations make easy to write types and signatures. Forms (b) and (c) contain an hidden kind variable, and are also correctly generated by the typechecker. Form (d) allows one to define a subtype as a list of cases from its supertype.

3.2 Use

Basic use

Use of polymorphic variants is identical to sum types, except that you don't have to define them. As a result, the same variant tag may be used in many different variant kinds.

```
# [`on; `off];;
- : [> off on] list = [`on; `off]
# `number 1;;
- : [> number(int)] = `number 1
# let f = function `on -> 1 | `off -> 0 | `number n -> n;;
val f : [< number(int) off on] -> int = <fun>
# List.map fun:f [`on; `off];;
- : int list = [1; 0]
# function `A -> `B | x -> x;;
- : 'a[A B .. > B] -> 'a[A B .. > B] = <fun>
```

`[> off on] list` means that to match this list, you should at least be able to match `` `off `` and `` `on ``, without argument. `[< number(int) off on]` means that `f` may be applied to `` `off ``, `` `on `` (both without argument), or `` `number `` *n* where *n* is an integer. Both variant kinds appearing only once in the type, the implicit type variables they constrain are not shown.

In the last example, since there is a relation between input and output (introduced by the `x -> x` case), the type variable appears. The variant kind `[A B .. > B]` means that `` `A `` and `` `B `` should have no argument, and any match on the output should at least have a case for `` `B ``.

Recursive variants are also correctly inferred.

```
# let rec map f = function `nil -> `nil
                 | `cons(a,l) -> `cons(f a, map f l);;
map : ('a -> 'b) ->
      'c[< cons('a * 'c) nil] -> 'd[> cons('b * 'd) nil] = <fun>
```

Another interesting use of variants is in combination with objects: rather than statically define a sum type to represent the object state, you may use variants.

```
# class switch = object
    val mutable state = (`off : [on off])
    method set = state <- `on
    method reset = state <- `off
    method get = state
  end;;
class switch :
  object
    val mutable state : [off on]
    method get : [off on]
    method reset : unit
    method set : unit
  end
```

This is more readable than using a boolean, and less cumbersome than defining a sum type. This is particularly true when you think that a sum type would have to be exported with the class, if you want to access it from other modules.

Subtleties

You can see in the previous example that we annotate `state` with the explicit type `[on off]`. This is necessary, since there cannot be free type variables in an object.

But even when this is not compulsory, this is good style to constrain variant types. Not constraining them may cause errors. For instance, the following code is typed correctly, but not in the expected way:

```
# let test state =
    if state = `on then print_endline "on"
    else print_endline "off";;
val test : [> on] -> unit = <fun>
```

Indeed, you probably expected the type `[on off] -> unit`. But there is nothing in this definition to know that. This problem only appears when either you use polymorphic equality, or an open form of pattern matching (when there is a case with no constructor).

As a result of these type annotations, and the use of type abbreviations, a polymorphic variant will often be less prolymorphic than it might. There are two ways to recover this lost polymorphism.

1. Variant dispatching: this relies on the use of `as` clauses in pattern matching. The variable you write after `as` will get the type of its associated pattern, rather than the type of the matched expression.

   ```
   # function `a|`b as x -> (Some x,None)
            | `a|`c as y -> (None,Some y);;
   - : [< a b c] -> [> a b] option * [> a c] option = <fun>
   ```

   This form of dispatching allows one to describe delegation behaviours in terms of variants, and provides an alternative to objects for small data.

2. Subtyping: like with object types, you may write explicit coercions for variant types. For instance, if x has type `[on off]`, you may extend it to type `[on off unknown]` by writing `(`x` :> [on off unknown])`. See next subsection for details.

Variants *vs.* **sum types**

Polymorphic variants and defined sum types are very similar. This may bother you when you are to choose the data structures to use in a program.

Advantages of polymorphic variants	Drawbacks/restrictions
No need to define them and to open signatures	In most cases you will still need to define an abbreviation for your comfort
Better accuracy of types / subtyping	Sometimes the type inferred is too general

For these reasons you will probably not use polymorphic variants for everything. In particular, sum types seem more appropriate for data structures shared throughout a program (when subtyping is not needed): you just have to group them in an header .mli file, opened from all other modules.

On the other hand, variants are appropriate for transient data structures. That is, simple data (flags, union types) used either locally or as parameter to functions. In particular, for functions called from out their defining module, using variants as parameters is cleaner: it avoids the grudge of prefixing arguments with the module name, and allows to share the same tags between different modules (particularly important if you are building a library, and do not want to use a C-like naming scheme).

Nothing opposes mixing both variants and sum types in the same data structure. Still, you shall be careful in making it easy to remember, by making clear what is defined as variant and what is defined as sum type.

3.3 Typing

A formal defintion of polymorphic variant typing in Objective Label can be found in [3]. We just explain here the basics. This supposes some knowledge of how works ML's type inference.

Variant kinds are an extension of the unification algorithm used by the typechecker. When you apply "`List.map` **fun**`:f : [< number(int) off on] list -> int list`" to "`[`on; `off] : [> off on] list`", the typechecker tries to computes the most general unifier[5], here `[< number(int) off on > off on]` and binds it to the variables which where constrained by each kind.

To compute this mgu ($\wedge$) we use the triple view (T,U,L) of variant kinds. When U is finite, T is restricted to its elements.

$$(T_1,U_1,L_1) \wedge (T_2,U_2,L_2) = ((T_1|_{U_1 \cap U_2}) \sqcap (T_2|_{U_1 \cap U_2}), U_1 \cap U_2, L_1 \cup L_2)$$

There are two logical conditions for this equality to stand:

1. $L_1 \cup L_2$ shall be included in $U_1 \cap U_2$.
2. $T_1 \sqcap T_2$ checks the compatibility between the two assigments by unifying their types recursively: if it fails there is no mgu.

Since subtyping is obtained through unification, it may happen that you unify two unrelated constrained variables. Like with objects, this can be avoided by explicit coercions. In `(`*expr* `:> [a b(int)])`, The type of *expr* is unified with `'a[< a b(int)]`, while the type of the result is `[a b(int)]`. For instance, to pass an argument `x : [off on]` to a

[5] In fact, the unification algorithm of O'Labl does not have the mgu property. That is, two kinds may have some subkind in common, and still not be unifiable directly: `[< none tag(int)]` and `[< none tag(char)]` for instance both contain `[none]`, but unification will fail for them. There is a slightly weaker system than the one implemented which recovers this property: if you don't discard disabled tags (those not appearing in the maximum list) when you unify, then `[none]` is no longer a proper subkind. It would be `[none tag(int) < none > none]` and `[none tag(char) < none > none]`, which take the same values but are different kinds. In practice, clearly we don't want to keep these disabled tags, and the actual algorithm can be seen as a refinement of the weaker algorithm, coercing types to enabled tags only.

function `f : [number(int) off on] -> int`, you should write `f (x :> [on off number('a)])`.

There is a small difference between class coercions and variant kind coercions, which appears on recursive types: while abbreviated class subtyping is recursive, this is not the case with kinds. In `(x :> 'a[nil cons(t * 'a)])`, the type of `x` will be unified with `[< nil cons(t * 'a[nil cons(t * 'a)])]` rather than with `'a[< nil cons(t * 'a)])` (which would correspond to classes). This just appears to be the right thing to do, since the kind of recursion you have with variants is different in nature from the recursion of objects. You can still coerce between recursive types by using complete coercions, of the form $(e : \tau_1 :> \tau_2)$.

Last, you shall never need to know about it, but another constraint is enforced by variant kind unification: it checks that all tags present in the result variant will be compiled to different integers[6]. This allows a very efficient compilation scheme, which makes polymorphic variants just as effective as sum types, as long as matches are of reasonable size: matching against a sum type works in constant time, but against polymorphic variants it is in $\log(n)$, where n is the number of cases (we cannot use the *switch* instruction), but the constant is lower.

4 Polymorphic methods

If you are using the Objective Caml class system, you certainly have bumped into this error message once:

```
# class c = object method m x = x end;;
Some type variables are unbound in this type:
  class c : object method m : 'a -> 'a end
The method m has type 'a -> 'a where 'a is unbound
```

And have thought "Why ?". Indeed, nothing opposes giving a polymorphic type to the method `m` in this example.

Still, the question is where this polymorphism shall occur. If we add `'a` as a type parameter parameter to the class, the class is polymorphic, but not the method itself.

```
# class ['a] c = object method m (x : 'a) = x end;;
class ['a] c : object method m : 'a -> 'a end
# let o = new c;;
val o : '_a c = <obj>
# o#m 1;;
- : int = 1
# o#m true;;
This expression has type bool but is here used with type int
```

Since all objects are the result of an application, they are always monomorphic, and their methods too.

In Objective Label you can explicitly declare a method to be polymorphic.

[6] The formula used to convert variant tag s of length n into a 31-bit integer is:

$$\mathsf{hash}(s) = \left(\sum_{i=1}^{n} s[i] * 223^{n-i}\right) \mathsf{mod}\ 2^{31}.$$

This means that all 4 character identifiers will be given different hash values. For more than 4 characters they may collide; but supposing that this functions gives us a random distribution, the probability $p(n)$ for two tags to collide in a n tag variant is (we use the average number of collisions):

$$p(n) \le \sum_{i=1}^{n-1} \frac{i}{2^{31}} \le \frac{n^2}{2^{32}}.$$

That is, with 256 tags, this probability is only $p(256) = 1.5 \cdot 10^{-5}$. Even $p(10000)$ is about 2 %.

```
# class c = object method m : 'a. 'a -> 'a = fun x -> x end;;
class c : object method m : 'a -> 'a end
# let o = new c;;
val o : c = <obj>
# o#m 1, o#m true;;
- : int * bool = 1, true
```

An explicitly polymorphic type is written *polytyp* ::= *typevar*+ . *typexpr*. It may only appear at 3 places:

- In a method definition, immediately after the method name.
- In a (virtual) method declaration.
- Inside object types: `< m : 'a. 'a -> 'a >`.

In class types and declarations (but not definitions), you can write either explicit or implicit polymorphic method types. When a type variable is not bound by any constraint, it will be automatically quantified universally[7].

```
class type id_spec = object method m : 'a -> 'a end
```

You can omit the type annotation on a method if it was previously declared polymorphic (in a super class, or by using virtual, or by annotating self).

```
# class c = object
    method virtual m : 'a. 'a -> 'a
    method m x = x
  end;;
class c : object method m : 'a -> 'a end
# class c = object (self : #id_spec)
    method m x = x
  end;;
class c : object method m : 'a -> 'a end
```

With the last method, do not forget the `#`: self's type has to be open. This method is particularly confortable since you can write implicitly polymorphic types.

As a limitation of the ML type system, polymorphic methods cannot be inferred. As a result, you can only use them if the object's type is already known. This can be made sure by an annotation.

```
# fun o -> o#m 1, o#m true;;
This expression has type bool but is here used with type int
# fun (o : c) -> o#m 1, o#m true;;
- : c -> int * bool = <fun>
```

The notion of "already known" is rigorously defined [4]. It is not enough for a type to be inferrable from the context, typing information must have been propagated with the data flow. See also remarks in subsection 2.5.

```
# fun o -> let _ = (o : c) in o#m 1, o#m true;;
Method m is not known as polymorphic
# fun o -> let o = (o : c) in o#m 1, o#m true;;
- : c -> int * bool = <fun>
```

In the first case, `o` is known to be of type `c`, but only by one of its uses. For the information to be fully valid, the typing information should be added to the one `o` we are going to use polymorphically.

Row variables can also be declared polymorphic. For this you must name them on their first occurence in the type.

[7] This behaviour introduces an incompatibility with O'Caml if you write constraints at the end of your class declaration. Always write them at the beginning.

```
# class c = object
    method m : 'a. (< m' : int; .. > as 'a) -> int = fun x -> x#m'
  end;;
class c = object method m : 'a. (< m' : int; .. > as 'a) -> int end
# class d = object
    method m : 'a. 'a[< a b] -> string = function `a -> "a" | `b -> "b"
  end;;
class d = object method m : 'a. 'a[< a b] -> string end
```

Again, in class types, writing the usual implicit polymorphic type is enough.

5 Other compiler changes

5.1 Record labels — NEW

The absence of overloading for record labels in Objective Caml has often been pointed out as a limitation, particularly for library development.

Objective Label tries to solve this by choosing the right label definition according to the typing environment. This is only possible when the type of the record is known (field extraction/modification) or the expected type is known (record creation). This feature is not supported for pattern matching.

```
# type point = { mutable x : int; y : int };;
type point = { mutable x: int; y : int }
# type box = { x : bool };;
type box = { x: bool }
# type 'a box = { x : 'a };;
type 'a box = { x: 'a }
# {x="hello"};;
- : string box = {x="hello"}
# {x=1;y=2};;
The label y belongs to the type point but is here mixed with labels of type
  int box
# let p : point = {x=1;y=2};;
val p : point = {x=1; y=2}
# p.x;;
- : int = 1
# p.x <- 3;;
- : unit = ()
```

This is an experimental feature. It may be suppressed in future versions.

5.2 Type information propagation

In general the order in which type information is propagated is left undefined. This is important to give a formal definition of the language indepent of a specific type inference algorithm.

However, there are cases where positive type information is needed to type-check a subexpression. Call to methods using labels or optionals in an object, or access to non-default fields in a record are two such cases.

In Objective Label there are 3 cases where type information propagation is guaranteed.

- let-defined identifiers propagate all the information given by their definition (bottom-up)
- if a function is defined with an expected type (top-down), its parameters propagate this information inside the function definition

- in function application, type information is propagated (top-down) from left to right

In practice this means that you may have to add type annotations when type information is not available, or change the order of parameters in a function application to make sure that type information is propagated.

```
# let l = [({x=1;y=2}: point);({x=3;y=4}: point)];;
val l : point list = [{x=1; y=2}; {x=3; y=4}]
# List.map fun:(fun r -> r.x) l;;
This expression has type point list but is here used with type 'a box list
# List.map fun:(fun (r : point) -> r.x) l;;
- : int list = [1; 3]
# List.map l fun:(fun r -> r.x);;
- : int list = [1; 3]
```

6 Distribution modifications

Modifications were done at several levels: library interfaces; parsing, type-checking and first stage of compilation; small modifications to other tools. Please refer to O'Caml's reference manual for how to use the commands.

6.1 Libraries

The standard library, as well as part of the other libraries of the distribution, are "labelized". This means that labels were added on function parameters to make clearer their meaning. We do not give the types here, you should either look at the compiled interface files (`.mli`) or use LablBrowser.

6.2 Batch compiler

`olablc` replaces `ocamlc`. Both commands are completely compatible and use same command line switches.

The bytecode files (`.cmo`) can be ran by the same `ocamlrun`, but one should take care of not mixing compiled interface files (`.cmi`): they are not compatible, and reading a compiled interface produced by one compiler with the other would result in a "xxx.cmi is not a compiled interface" error.

Command line switches were added.

`-nolabels` makes `olablc` source-compatible with `ocamlc`. All labels are ignored: no parameter reordering is done at compilation. Optional arguments of type *t* become arguments of type *t* `option`. Coherence of labeling with the interface (`.mli`) is not verified. This option is the default if you rename the compiler to `ocamlc`.

`-labels` makes the compiler behave as `olablc`. This is only useful if you renamed `olablc` in `ocamlc`.

`-w K/k` was added to turn the "unknown method" warning on and off. The default warnings are now `FKMPsUVX`, that is all warnings except non-unit statements.

By using the `-nolabels` switch or renaming the compiler, all O'Caml programs and libraries can be compiled and used in O'Labl[8]. They can be easily "labelized" by adding labels only in the interface: they are ignored when compiling the implementation, but can be used in O'Labl programs. For bootstrapping reasons, this is the way the standard library was labelized. This explains why it contains no optional arguments.

[8] To give an idea of the degree of compatibility when using `-nolabels`, the browser MMM can be compiled by O'Labl without any modification.

If you compile and install the native code compiler, `olablopt` will also replace `ocamlopt`. (*cf.* O'Caml's reference manual for differences between bytecode and native code compilers.)

6.3 Interactive toplevel

`olabl` replaces `ocaml`. Both commands are completely compatible and use same command line switches.

`-nolabels` is also available with `olabl`. This allows one to read O'Caml files in the toplevel, replacing completely `ocaml`. Again, you obtain the same effect by renaming `olabl` to `ocaml`.

Among other interesting switches, notice also `-dlambda`. This dumps the code after label compiling, allowing to compare labeled code with usual one.

Custom toplevels can be built using `olablmktop`.

6.4 Other tools

They are all available under names starting with `olabl`. `olablcp`, `olablprof` and `olabldebug` needed small changes for use with Objective Label, but `olablrun`, `olabllex`, `olablyacc` and `olabldep` are identical to their Objective Caml counterparts.

7 LablTk

LablTk41 is an easy-to-use and well integrated interface to the Tcl/Tk toolkit. It is more than the labelized version of OCamlTk41, the Tcl/Tk interface of O'Caml. Optional arguments as well as polymorphic variants were heavily used to make it much closer to Tcl's syntax, and almost type safe. See the (rather old) documentation for details, and read the examples for how to use it. You can then look for needed functions using LablBrowser.

Commands available to use LablTk are:

- `labltk` An interactive toplevel.
- `labltklink` A bytecode compiler/linker configured for use with LablTk.
- `labltkopt` Native version of `labltklink`.

These three commands are just shell scripts. For fancier uses, you can look at the path and libraries they are setting.

8 LablBrowser

LablBrowser is a source and compiled interface browser, written using LablTk. This is a necessary companion to the programmer.

Its functions are:

- navigation through O'Labl's modules (using compiled interfaces).
- source editing, type-checking, and browsing.
- integrated O'Labl shell, running as a subprocess.

The name of the command is `lablbrowser`, and, as with the compiler, you may change the standard library directory by setting `LABLLIB`, and extend the load path using `-I` options.

8.1 Viewer

This is the first window you get when you start LablBrowser. It displays the list of modules in the load path. Click on one to start your trip.

- The entry line at the bottom allows one to search for an identifier in all modules (wildcards "?" and "*" allowed). If the input contains an arrow ("–>"), the search is done by type inclusion (*cf.* Search Symbol - Included type).
- The **Close all** button is there to dismiss the windows creating during you trip (every click creates one. . .) By double-clicking on it you will quit the browser.
- **File - Open** and **File - Editor** give access to the editor.
- **File - Shell** creates an O'Labl subprocess in a shell.
- **Modules - Path editor** changes the load path. **Modules - Reset cache** rescans the load path and resets the module cache. Do it if you recompile some interface, or get confused about what is in the cache.
- **Modules - Search symbol** allows to search a symbol either by its name, like the bottom line of the viewer, or, more interestingly, by its type. **Exact type** searches for a type with exactly the same information as the pattern (variables match only variables). **Included type** allows to give only partial information: the actual type may take more arguments and return more results, and variables in the pattern match anything. In both cases, argument and tuple order is irrelevant[9], and unlabeled arguments in the pattern match any label.

8.2 Module walking

Each module is displayed in its own window.

At the top, a scrollable list of the defined identifiers. If you click on one, this will either create a new window (if this is a sub-module) or display the signature for this identifier below.

Signatures are clickable. Double clicking with the left mouse button on an identifier in a signature brings you to its signature, inside its module box.

A single click on the right button pops up a menu displaying the type declaration for the selected identifier. Its title, when selectable, also brings you to its signature.

At the bottom, a series of buttons, depending on the context.

- **Show all** displays the signature of the whole module.
- **Detach** copies the currently displayed signature in a new window, to keep it.
- **Impl** and **Intf** bring you to the implementation or interface of the currently displayed signature, if it is available.

C-s lets you search a string in the signature.

8.3 File editor

You can edit files with it, if you like to live dangerously. Otherwise you can use it as a browser, making occasional corrections.

The **Edit** menu contains commands for jump (C-g), search (C-s), and sending the current selection to a sub-shell (M-x). For this last option, you may choose the shell via a dialog.

Essential functions are in the **Compiler** menu.

[9]To avoid combinatorial explosion of the search space, optional arguments in the actual type are ignored in the actual if (1) there are too many of them, and (2) they do not appear explicitly in the pattern.

- **Preferences** opens a dialog to set internals of the editor and type-checker.
- **Lex** adds colors according to lexical categories.
- **Typecheck** verifies typing, and memorizes to let one see an expression's type by double-clicking on it. This is also valid for interfaces. If an error occurs, the part of the interface preceding the error is computed.

  After typechecking, pressing the right button pops up a menu giving the type of the pointed expression, and eventually allowing to follow some links.
- **Clear errors** dismisses type-checker error messages and warnings.
- **Signature** shows the signature of the current file (after type checking).

8.4 Shell

When you create a shell, a dialog is presented to you, letting you choose which command you want to run, and the title of the shell (to choose it in the Editor).

You may change the default command by setting the `OLABL` environment variable.

The executed subshell is given the current load path.

- **File** use a source file or load a bytecode file. You may also import the browser's path into the subprocess.
- **History** M-p and M-n browse up and down.
- **Signal** C-c interrupts, and you can also kill the subprocess.

8.5 Bugs

- This not really a bug, but LablBrowser is a huge memory consumer. Go and buy some.
- When you quit the editor and some file was modified, a dialogue is displayed asking whether you want to really quit or not. But 1) if you quit directly from the viewer, there is no dialogue at all, and 2) if you close from the window manager, the dialogue is displayed, but you cannot cancel the destruction... Beware.
- When you run it through xon, the shell hangs at the first error. But its ok if you start lablbrowser from a remote shell...

Acknowledgements

We have to thank very heartfully the Caml team. All these extensions were possible thanks to the O'Caml/CamlTk system.

A Installation

Using the new Objective Label distribution, installation shall be very easy. We just reproduce here the procedure described in the INSTALL file.

1. Copy `ocaml-2.04.tar.gz` to the directory `olabl-2.04`.
2. Configure the Makefile in this directory.
3. Make sure that you patch program is good enough, by checking `patch -v`. The installation procedure was only checked for GNU `Patch version 2.1`.

4. If you are sure enough of your configuration, just type `make world-install` or `make worldopt-install`. What they will do is:

   ```
   % make patch
   % make configure
   % make olabl (or olablopt)
   % make install-olabl
   % make libs (or libsopt)
   % make install-libs
   ```

   Otherwise type `make`, and follow the instructions.

5. The emacs mode (`ocaml-2.04/emacs`) and LablTk examples (`labltk41/example`) are not automatically compiled. You must go to their directories and compile/install by hand.

6. You can now use LablBrowser towards a brand new labeled world...

   ```
   % lablbrowser
   ```

Here is were installed files live.

- olablc, olablopt, olabl, olablmktop, olablrun, olabllex, olablyacc, olabldep, olablcp, olablprof and olabldebug are installed in `BINDIR`.
- Libraries are installed in `LIBDIR`, and manpages in `MANDIR`.
- For LablTk, commands (labltk, labltklink, labltkopt, lablbrowser) are installed in `BINDIR` and libraries in `LIBDIR/labltk41`.

If you do not plan to install O'Caml, but want to be able to compile O'Caml programs easily, copy or link `olablc` and `olabl` to `ocamlc` and `ocaml`: with these names, they will work in compatibility mode. You may also copy or make links for other utilities.

References

[1] Hassan Aït-Kaci and Jacques Garrigue. Label-selective λ-calculus: Syntax and confluence. *Theoretical Computer Science*, 151:353–383, 1995.

[2] Jun P. Furuse and Jacques Garrigue. A label-selective lambda-calculus with optional arguments and its compilation method. RIMS Preprint 1041, Research Institute for Mathematical Sciences, Kyoto University, October 1995.

[3] Jacques Garrigue. Programming with polymorphic variants. In *ML Workshop*, September 1998.

[4] Jacques Garrigue and Didier Rémy. Extending ML with semi-explicit higher-order polymorphism. In Abadi and Ito, editors, *Proc. of the International Conference on Theoretical Aspects of Computer Software*, number 1281 in Springer-Verlag LNCS, pages 20–46, Sendai, Japan, September 1997.

[5] François Rouaix. *The CamlTk Interface*. INRIA, Rocquencourt, France, July 1995. Available at URL http://pauillac.inria.fr/camltk.