Labeled and optional arguments for Objective Caml*

Jacques Garrigue†

Abstract

We added labeled and optional arguments to the Objective Caml language, combining currying and commutation. Contrary to a previous attempt, the formal system we present here defines semantics of both out-of-order application and optional parameter discarding independently of types. We extend the ML type system to cope with these new features, and show how types can be used to obtain an efficient compilation method.

1 Introduction

There is a huge gap between computer language syntax and natural language. While both can be seen as sequences of sentences —definitions, statements and questions— held together by a rather large set of connectives, the freedom in how to write these sentences is almost nil in programming languages.

Stylistic freedom, and ability to omit some implicit parts of a sentence is sometimes shunned as sloppiness. However, anybody who writes often enough knows that style matters, and that even omitting implicit facts may make your exposure clearer.

Where we should bother about sloppiness is in the semantics, and in the way people actually write programs, but not in what they are allowed to write. Certainly we do not want to trade away a simple and well defined semantics for a complex and ambiguous one. In this respect overloading is not an universal solution. While there is an argument about overloading being omnipresent in natural language, programming languages intend to avoid ambiguities, and overloading very soon creates some.

By introducing labeled arguments in the syntax, we are able to propose a mechanism which provides flexibility in the function call syntax, allowing various parameter orders [AKG95, GAK94], and the omission of implicit parameters. Yet, it does so in a purely syntactic manner: types, and overloading, do not intervene in the resolution of the function call. One can still understand a program as a formally evaluated expression, that resolves to a result on the sole basis of its data contents. This is similar to named parameters in Common Lisp [Ste84], and to a lesser degree call patterns in Smalltalk [GR83], but we add types and currying. Inferred labeled types annotate expressions, without affecting their evaluation. Types play an important role in compilation, allowing one to produce efficient code, which respects the semantics while computing in a different, more efficient manner.

This mechanism, which departs slightly from the one designed by Jun Furuse [FG95] and used in Objective Label [Gar99], is now included in the Objective Caml language [LDG+00]. The main differences with the previous formalism are that resolution of optional arguments does not rely on types anymore, and full compliance with the call-by-value semantics.

From a technical point of view, the contribution of this paper is a reduction system combining currying, labeled and optional arguments. Works by others [Lam88, Dam98] did not consider the full combination of currying and commutation (some kind of non-commuting closure operation is needed between passing groups of parameters), and the Objective Label approach had to rely on types for optional parameters. More practically, we give here the first formal account of the semantics of labels in Objective Caml 3, hoping that this will help improve the understanding of this feature.

---

*Revised version of 2000-04-12, there was a bug in reduction rules.

†Research Institute for Mathematical Sciences, Kyoto University, Kitashirakawa-oiwakecho, KYOTO 606-8502. E-mail: `garrigue@kurims.kyoto-u.ac.jp`. Web: `http://wwwfun.kurims.kyoto-u.ac.jp/~garrigue/`.

After giving a few examples of the intended uses, we give a formal description of the system, of its typing, and explain how it can be compiled efficiently. We also shortly describe an alternative semantics, intended to make mixing of labeled and non-labeled code easier.

2 Labels at work

Originally labels were introduced in Objective Label's libraries as proof-of-concept experiment. However they fast appeared to be both useful and comfortable. This feeling of comfort may be surprising, since contrary to what happens in Ada [Led81], a language not specially known for its conciseness, labels in Objective Label are a compulsory feature. That is, if a function makes use of labeled arguments, then these arguments must be imperatively labeled at every call site.

Here are the types of some functions of the Objective Caml 3.00 standard library.

```
output : out_channel -> buf:string -> pos:int -> len:int -> unit
Hashtbl.add : ('a, 'b) Hashtbl.t -> key:'a -> data:'b -> unit
List.map : f:('a -> 'b) -> 'a list -> 'b list
```

Non-optional labels appear in types as *label*:, just before the type of the labeled argument.

This kind of compulsory labels have two goals: to disambiguate the role of an argument, and to allow commutation between parameters. The first one improves readability of programs by avoiding having to refer to external documentation, and the second also improves readability, by giving more freedom in the layout of programs. Commutation is only available when the Objective Caml interpreter is started with the `-labels` option.

```
# output stdout ~buf:"Hello world" ~pos:0 ~len:5;;
Hello- : unit = ()
# List.map ["a";"b"] ~f:(fun s -> "This is '" ^ s ^ "'");;
- : string list = ["This is 'a'"; "This is 'b'"]
# Hashtbl.add ~key:5;;
- : (int, 'a) Hashtbl.t -> data:'a -> unit = <fun>
```

In function applications, labels appear as ~*label*:, to avoid confusion with type annotations. Partial applications are allowed: arguments which were not given are kept abstract.

One can also define functions taking labeled arguments, both compulsory and optional (prefixed by "?"). Optional parameters bind their variable to `Some v` if an actual value was passed, and `None` if it was omitted. Internally an optional parameter is of type τ `option` = `Some of` τ | `None`.

```
# let rec gcd ~m ~n =
#   if n = 0 then m else gcd ~m:n ~n:(m mod n);;
val gcd : m:int -> n:int -> int = <fun>
# let exp ?base x =
#   let base = match base with None -> 2.0 | Some b -> b in
#   base ** x;;
val exp : ?base:float -> float -> float = <fun>
# exp 3.0;;
- : float = 8.000000
# exp ~base:3.0 3.0;;
- : float = 27.000000
# exp  3.0 ~base:3.0;;
- : float = 27.000000
# let exp' ?(base = 2.0) x = base ** x;;
val exp' : ?base:float -> float -> float = <fun>
```

Since in most cases label and variable share the same name, we use ~*name* as an abbreviation for ~*name*:*name*, and similarly ?*name* is ?*name*:*name*. The basic rule for discarding an optional parameter is to omit it when a non-labeled parameter appearing after it is passed.

[Beta]
$(\ldots((\texttt{fun}~\tilde{}\mathrm{l_i}{:}x \to e)~\tilde{}\mathrm{l_1}{:}e_1 \ldots) \ldots \tilde{}\mathrm{l_i}{:}e_i \ldots \tilde{}\mathrm{l_n}{:}e_n)$ when $\mathrm{l_i} \notin \{\mathrm{l_1} \ldots \mathrm{l_{i-1}}\}$

$\to (\ldots(e[e_i/x]~\tilde{}\mathrm{l_1}{:}e_1 \ldots) \ldots \tilde{}\mathrm{l_{i-1}}{:}e_{i-1}~\tilde{}\mathrm{l_{i+1}}{:}e_{i+1} \ldots \tilde{}\mathrm{l_n}{:}e_n)$

[Beta-Some]
$((\texttt{fun}~?\mathrm{l_i}{:}x \to e)~\tilde{}\mathrm{l_1}{:}e_1 \ldots \tilde{}\mathrm{l_n}{:}e_n)$ when $\mathrm{l_i} \notin \{\mathrm{l_1} \ldots \mathrm{l_{i-1}}\}$

$\to (e[\mathsf{Some}(e_i)/x]~\tilde{}\mathrm{l_1}{:}e_1 \ldots \tilde{}\mathrm{l_{i-1}}{:}e_{i-1}~\tilde{}\mathrm{l_{i+1}}{:}e_{i+1} \ldots \tilde{}\mathrm{l_n}{:}e_n)$

[Beta-None] when $\mathrm{l_i} = \emptyset$ and $\mathrm{l} \notin \{\mathrm{l_1} \ldots \mathrm{l_n}\}$
$((\texttt{fun}~?\mathrm{l}{:}x \to e)~\tilde{}\mathrm{l_1}{:}e_1 \ldots \tilde{}\mathrm{l_n}{:}e_n) \to (e[\mathsf{None}/x]~\tilde{}\mathrm{l_1}{:}e_1 \ldots \tilde{}\mathrm{l_n}{:}e_n)$

[Merge] when $\emptyset \notin \{\mathrm{l_1} \ldots \mathrm{l_m}\}$
$((e~\tilde{}\mathrm{l_1}{:}e_1 \ldots \tilde{}\mathrm{l_m}{:}e_m)~\tilde{}\mathrm{l_{m+1}}{:}e_{m+1} \ldots \tilde{}\mathrm{l_n}{:}e_n) \to (e~\tilde{}\mathrm{l_1}{:}e_1 \ldots \tilde{}\mathrm{l_n}{:}e_n)$

[Let] $\texttt{let}~x = e_1~\texttt{in}~e_2 \to e_2[e_1/x]$

Figure 1: Reduction rules

Note that this requires considering application as a n-ary operation, since we still want to be able to commute optional and non-labeled arguments in the same function call. Some syntactic sugar is available to give a default value to optional arguments: the definition of `exp'` expands to the same code as `exp`. Notice that optional parameters are stronger than parameters with a fixed default value: the default value may be computed, or we may choose to do something completely different according to the call pattern.

In practice optional parameters are often used to pass attribute values, and can be very numerous. Here is the type of the frame creation function in LablTk, the idea being to have the most natural syntax: attributes are just parameters to the widget creation call, like they would probably be in non-statically typed languages like Lisp, Tcl or a command-line shell.

```
Frame.create :
  ?name:string -> ?background:Tk.color -> ?borderwidth:int ->
  ?clas:string -> ?colormap:Tk.colormap -> ?cursor:Tk.cursor ->
  ?height:int ->  ?highlightbackground:Tk.color ->
  ?highlightcolor:Tk.color ->  ?highlightthickness:int ->
  ?relief:Tk.relief -> ?takefocus:bool ->  ?visual:Tk.visual ->
  ?width:int -> 'a Widget.widget -> Widget.frame Widget.widget
```

A more detailed presentation of labeled and optional arguments can be found in the tutorial part of [LDG+00].

3 Syntax and reduction semantics

We define a simplified version of the syntax. It departs from traditional lambda-calculus with let-polymorphism by the introduction of labels (both optional and non-optional) in abstractions, and the fact that application is n-ary (and labeled).

l	::=	$\emptyset \mid$ <*name*>	labels
e	::=	x	variable
	$\mid$	$\texttt{fun}~\tilde{}\mathrm{l}{:}x \to e$	labeled abstraction
	$\mid$	$\texttt{fun}~?\mathrm{l}{:}x \to e$	optional abstraction
	$\mid$	$\texttt{let}~x = e~\texttt{in}~e$	polymorphic let
	$\mid$	$(e~\tilde{}\mathrm{l}{:}e \ldots \tilde{}\mathrm{l}{:}e)$	n-ary application

Non-labeled argument are just handled as arguments labeled by $\emptyset$.

In figure 1, we need to replace β by 3 reduction rules: normal case, optional present case, and optional absent case. The [Merge] rule has only an administrative role. The last rule is usual let-reduction.

[Beta] may look complex, because it preserves the multi-application structure, but all it does is selecting the first argument with the right label. Without multi-application, we can

$$\Gamma, x : \forall \alpha_1 ... \alpha_n.\tau \vdash x : \tau[\tau_1 ... \tau_n / \alpha_1 ... \alpha_n]$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau}{\Gamma \vdash \texttt{fun}\ \tilde{\mathrm{l}}{:}x \to e : \mathrm{l}{:}\tau_1 \to \tau} \qquad \frac{\Gamma, x : \tau_1\ \texttt{option} \vdash e : \tau}{\Gamma \vdash \texttt{fun}\ ?\mathrm{l}{:}x \to e :\ ?\mathrm{l}{:}\tau_1 \to \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \forall \mathsf{FTV}(\tau_1) \setminus \mathsf{FTV}(\Gamma).\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_0 : {}^{\mathrm{o}_1}\mathrm{l}_1{:}\tau_1 \to \ldots \to {}^{\mathrm{o}_\mathrm{n}}\mathrm{l}_\mathrm{n}{:}\tau_n \to \tau_0 \qquad \Gamma \vdash e_1 : \tau_{\sigma_1} \quad \ldots \quad \Gamma \vdash e_m : \tau_{\sigma_m}}{\Gamma \vdash (e_0\ \mathrm{l}_{\sigma_1}{:}e_1 \ldots \mathrm{l}_{\sigma_\mathrm{m}}{:}e_m) : \mathsf{erase}_k({}^{\mathrm{o}_{\sigma_{\mathrm{m}+1}}}\mathrm{l}_{\sigma_{\mathrm{m}+1}}{:}\tau_{\sigma_{m+1}} \to \ldots \to {}^{\mathrm{o}_{\sigma_\mathrm{n}}}\mathrm{l}_{\sigma_\mathrm{n}}{:}\tau_{\sigma_n} \to \tau_0)}$$

o_i is an optional mark, that is either nothing or "?".
σ is a permutation of $[1, n]$ such that (1) a label does not commute with itself: if $i < j$ and $l_i = l_j$ then $\sigma_i < \sigma_j$, and (2) extra parameters are kept in order: if $i < j$ and $\sigma_i > m$ and $\sigma_j > m$ then $\sigma_i < \sigma_j$.
k is the number of parameters remaining that appeared in the type before a non labeled argument that was passed: $|\{i \in [1, n] \mid \sigma_i > m \wedge (\exists j > i)\ \mathrm{l_j} = \emptyset \wedge \sigma_j \leq m\}|$.

$\mathsf{erase}_k(\tau)$ is inductively defined as $\begin{cases} \mathsf{erase}_0(\tau) & = \quad \tau \\ \mathsf{erase}_{k+1}(?\mathrm{l}{:}\tau_1 \to \tau) & = \quad \mathsf{erase}_k(\tau) \\ \mathsf{erase}_{k+1}(\mathrm{l}{:}\tau_1 \to \tau) & = \quad \mathrm{l}{:}\tau_1 \to \mathsf{erase}_k(\tau) \end{cases}$

Figure 2: Typing rules

write it in a much simpler form:

$$(\texttt{fun}\ \tilde{\mathrm{l}}_\mathrm{n}{:}x \to e)\ \tilde{\mathrm{l}}_1{:}e_1 \ldots \tilde{\mathrm{l}}_\mathrm{n}{:}e_n \to e[e_n/x]\ \tilde{\mathrm{l}}_1{:}e_1 \ldots \tilde{\mathrm{l}}_{\mathrm{n}-1}{:}e_{n-1}$$

Multi-application structure is needed for optional arguments. [Beta-Some] does basically the same thing as [Beta], selecting the first occurrence of an argument with label $\mathrm{l_i}$, and substituting the argument `Some` e_i for x in the body of the function, but l_i is restricted to appear in the first multi-application. Symmetrically, [Beta-None] handles the case when there is no argument with label l, but there is a non-labeled argument, meaning that the parameter was omitted; then `None` is substituted for x.

If there is no non-labeled argument in a multi-application, then it may be merged with the next englobing multi-application, with [Merge]. This is coherent with the [Beta-Some] and [Beta-None] rules, and gives them new chances to apply.

4 Typing

Types are the usual monotypes and polytypes of the Hindley-Milner polymorphic type system. The arrow is labeled, and split in two cases: non-optional and optional label.

$$\begin{array}{lll} \tau & ::= & \alpha \mid \tilde{\mathrm{l}}{:}\tau \to \tau \mid ?\mathrm{l}{:}\tau \to \tau \\ \sigma & ::= & \forall \alpha ... \alpha.\tau \end{array}$$

Typing rules are given in figure 2. The two abstraction rules reflect labels in the syntax at the type level. The application rule looks complex, essentially because it integrates all the commuting and discarding machinery.

Contrary to [FG95], commutation is not integrated here as a congruence on types, but as a permutation σ which allows a reordering of parameters during application. The handling of optional arguments differs also: $\mathsf{erase}_k(\tau)$ matches the untyped rules, rather than being a definition by itself.

Theorem 1 (Subject Reduction) *If* $\Gamma \vdash e : \tau$ *and* $e \to e'$, *then* $\Gamma \vdash e' : \tau$.

While we have the subject reduction property, our system is weaker than Hindley-Milner, in that it doesn't enjoy the principal type property, an immediate consequence being the absence of complete type inference algorithm. On the other hand, there is a straightforward partial inference algorithm, and experience shows that type annotations are almost never required: it is enough to be able to use out-of-order application and optional arguments on known functions. On unknown functions, the partial algorithm may assume a wrong type, and fail later:

```
# let f g = g ~y:0 ~x:1;;
val f : (y:int -> x:int -> 'a) -> 'a = <fun>
# f (fun ~x ~y -> x - y);;
This function should have type y:int -> x:int -> 'a
but its first argument is labeled ~x
```

In [FG95], we attempted to solve this problem by allowing label commutation in some types, but the way it was compiled created potential gaps with call-by-value semantics (it was correct in a call-by-name setting), and moreover we could not allow commutation in more complex cases, like the types of object methods, or when optional arguments are implied.

5 Call-by-value evaluation

In order to define a precise notion of call-by-value evaluation for this calculus, we need to define what are values and evaluation contexts. In turn, this formal definition of call-by-value will tell us what the compiler should preserve of the reduction semantics.

$$\begin{array}{rcll} v & ::= & x \mid \mathtt{fun}\ \tilde{}\mathrm{l}{:}x \to e \mid \mathtt{fun}\ ?\mathrm{l}{:}x \to e & \text{head-normal} \\ & \mid & (\ldots((\mathtt{fun}\ \tilde{}\mathrm{l}{:}x \to e)\ \tilde{}\mathrm{l}_1{:}v \ldots)\ldots \tilde{}\mathrm{l}_n{:}v) & \text{when } \mathrm{l} \notin \{\mathrm{l}_1,\ldots,\mathrm{l}_n\} \\ & \mid & ((\mathtt{fun}\ ?\mathrm{l}{:}x \to e)\ \tilde{}\mathrm{l}_1{:}v \ldots \tilde{}\mathrm{l}_n{:}v) & \text{when } \emptyset \notin \{\mathrm{l}_1,\ldots,\mathrm{l}_n\} \end{array}$$

While the reason for making application of a normal abstraction on differing labels a value is clear —there is no β-rule to reduce this term—, the last case is more surprising, since there are actually rules to reduce this instance of application (whenever $\mathrm{l} \in \{\mathrm{l}_1 \ldots \mathrm{l}_n\}$). This just means that when such an application will be evaluated is left undefined: it may happen sooner, later, or never[1]. The reason we want to allow delaying the evaluation of such a redex will be explained in section 6. We would obtain the same freedom by assuming that no side-effect is caused until the residual of this term is applied to a non-labeled argument, which sounds like a reasonable assumption, but this cannot be easily enforced.

$$\begin{array}{rcll} E & ::= & [\,] & \text{hole} \\ & \mid & \mathtt{let}\ x = E\ \mathtt{in}\ e & \text{let-bound value} \\ & \mid & (E\ \tilde{}\mathrm{l}{:}e \ldots \tilde{}\mathrm{l}{:}e) & \text{application head} \\ & \mid & (e\ \tilde{}\mathrm{l}{:}e \ldots \tilde{}\mathrm{l}{:}E \ldots \tilde{}\mathrm{l}{:}e) & \text{application argument} \end{array}$$

Again evaluation contexts do not specify in which order the different parts of an application should be evaluated. We can even evaluate an argument before the function itself is evaluated.

Finally, in call-by-value semantics, reduction rules only apply when a redex composed only of values appears in an evaluation context.

$$[\mathsf{Beta}_v]\ \frac{(v\ \tilde{}\mathrm{l}_1{:}v_1 \ldots \tilde{}\mathrm{l}_n{:}v_n) \to e}{E[(v\ \tilde{}\mathrm{l}_1{:}v_1 \ldots \tilde{}\mathrm{l}_n{:}v_n)] \to_v E[e]} \qquad [\mathsf{Let}_v]\ E[\mathtt{let}\ x = v\ \mathtt{in}\ e] \to_v E[e[v/x]]$$

Now we can define call-by-value reduction $\Rightarrow_v$ as the repeated rewriting of the root of a term e, leading to a value v.

$$e \to_v e_1 \to_v \ldots \to_v e_n \to_v v$$

Since we allow some freedom in the semantics, there may actually be several paths from e to v, and in the presence of side-effects the final result v may vary. An evaluator or a compiler is correct if it is sound —any result it computes can be derived by the call-by-value reduction—, and partially complete —it may be non-terminating only if there is an infinite call-by-value reduction path.

[1] We might actually fully define it by adding the symmetrical constraint to [Merge-Opt], but not doing so leaves more freedom to the compiler. We just follow here Objective Caml's design, in which the evaluation order of the parts of an application redex is not specified.

6 Compilation

Compilation is type-directed, and of course correct with respect to the call-by-value semantics. We do not describe it formally here, since it is straightforward, thanks to the strict correspondence between terms and types. Let's just see how it works on a few examples. If f has type pos:`int` $\rightarrow$?len:`int` $\rightarrow$ `string` $\rightarrow$ `string`, here are some labeled applications and their translations.

Labeled	Translation
`(f "Hello" ~len:2 ~pos:3)`	f 3 (`Some` 2) `"Hello"`
`(f "Hello" ~pos:3)`	f 3 `None` `"Hello"`
`(f ~len:(5-2))`	`let` $len = 5 - 2$ `in fun` $pos \rightarrow f\ pos$ (`Some` len)
`(f "Hello")`	`fun` $pos \rightarrow f\ pos$ `None` `"Hello"`

The first case is the "normal" one: all arguments are given, in some arbitrary order. Since we have the type of the function, we can pass them in the proper order. Notice also that since the order of evaluation of arguments is not specified anyway, we do not need to take any special care for preserving semantics in this case.

The second case is an omitted argument. It is silently replaced by `None`. Again a labeled application is just compiled into an unlabeled one.

The third and fourth cases are out-of-order applications. They require eta-expansion to be compiled correctly. We must be careful about not changing the order of evaluation: arguments must be computed before building the closure.

All the above examples were optimal in their compiled form: this is exactly what one would write by hand, if labeled arguments were not available. Now let's consider two functions g : pos:`int` $\rightarrow$ len:`int` $\rightarrow$ `string` $\rightarrow$ `string` and h : ?pos:`int` $\rightarrow$?len:`int` $\rightarrow$?stride:`int` $\rightarrow$ `string` $\rightarrow$ `string`.

Labeled	Translation
`(g "Hello")`	`fun` $pos \rightarrow$ `let` $g_1 = f\ pos$ `in fun` $len \rightarrow g_1\ len$ `"Hello"`
`(h ~stride:2)`	`fun` $pos \rightarrow$ `fun` $len \rightarrow h\ pos\ len$ (`Some` 2)

Applying g is somehow less than optimal: we build a closure g_1 by partially applying it on pos, to be sure that we conform with the call-by-value semantics, and have all side-effects in the expected order. Yet there is a high probability that such a partial application is not side-effecting anyway, so that this costly code is not needed. Luckily, you have rarely enough parameters in a function for it to be a real problem.

But, as seen in section 2, a function may have lots of optional parameters. Were you to partially apply on the last of them, and (taking `Frame.create` as example) you would end up building 13 useless closures in a row! This is why applying a function on an optional label is handled as a value in the semantics: then you are free to delay evaluation until application on a non-labeled parameter forces it. This is what is done with h. A strict semantics would have required a closure here also, but simple eta-expansion is enough with our weaker notion of value.

7 Alternative semantics

The semantics we defined above form a conservative extension of the original Caml semantics. However, when integrating these features in an existing system, the question of libraries is paramount. If they are left without labels, then the use of labels would be restricted to new libraries, loosing the benefits of having them in the standard library. On the other hand, adding labels to the standard library would break source code compatibility.

After considering various alternative, including using two sets of standard libraries, we settled for having yet another semantics: the classic mode. This is actually the default mode of the Objective Caml system. It trades commutation of non-optional arguments for the ability to omit labels. Thanks to this, labels can be freely added to libraries, without breaking compatibility with legacy code. Yet, this mode is fully functional, and you can both use labels (to catch some errors) and optional arguments in programs.

[Beta]

$(\texttt{fun}\ {\sim}\mathsf{l}{:}x \to e)\ {\sim}\mathsf{l}'{:}e' \to e[e'/x]$

[Beta-Some] when $\{\mathsf{l}, \emptyset\} \not\cap \{\mathsf{l}_1 \ldots \mathsf{l}_n\}$

$(\texttt{fun}\ ?\mathsf{l}{:}x \to e)\ {\sim}\mathsf{l}_1{:}e_1 \ldots {\sim}\mathsf{l}_n{:}e_n\ {\sim}\mathsf{l}{:}e' \to e[\texttt{Some}(e')/x]\ {\sim}\mathsf{l}_1{:}e_1 \ldots {\sim}\mathsf{l}_n{:}e_n$

[Beta-None] when $\mathsf{l} \notin \{\mathsf{l}_1 \ldots \mathsf{l}_n\}$

$(\texttt{fun}\ ?\mathsf{l}{:}x \to e)\ {\sim}\mathsf{l}_1{:}e_1 \ldots {\sim}\mathsf{l}_n{:}e_n\ {\sim}\emptyset{:}e' \to e[\texttt{None}/x]\ {\sim}\mathsf{l}_1{:}e_1 \ldots {\sim}\mathsf{l}_n{:}e_n\ {\sim}\emptyset{:}e'$

Figure 3: Classic mode reduction rules

$$\frac{\Gamma \vdash e : \mathsf{l}{:}\tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\ {\sim}\mathsf{l}'{:}e' : \tau}$$

$$\frac{\Gamma \vdash e : ?\mathsf{l}_1{:}\tau_1 \to \ldots \to ?\mathsf{l}_n{:}\tau_n \to ?\mathsf{l}{:}\tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\ {\sim}\mathsf{l}{:}e' : ?\mathsf{l}_1{:}\tau_1 \to \ldots \to ?\mathsf{l}_n{:}\tau_n \to \tau}\ \mathsf{l} \notin \{\mathsf{l}_1 \ldots \mathsf{l}_n\}$$

$$\frac{\Gamma \vdash e : ?\mathsf{l}_1{:}\tau_1 \to \ldots \to ?\mathsf{l}_n{:}\tau_n \to \mathsf{l}{:}\tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\ e' : \tau}$$

Figure 4: Classic mode typing for application

Expressions are the same, but since optional parameters can no longer commute with non-optional ones, we can consider application as binary.

$$e ::= \ldots \mid e\ {\sim}\mathsf{l}{:}e \mid e\ ?\mathsf{l}{:}e$$

Reduction rules are given in figure 3, plus the original [Let] rule. In [Beta], labels are simply ignored, and reduction progresses independently of them. [Beta-Some] and [Beta-None] are simplifications of the original rules, since we need not bother about commutation. Administrative rules are not needed anymore.

Typing differs in spirit: we add a congruence on types, so that non-optional labels can be safely ignored.

$$\mathsf{l}{:}\tau \to \tau' \simeq \mathsf{l}'{:}\tau \to \tau'$$

This congruence applies anywhere inside a type, but it does not apply to optional labels. Typing rules are those of figure 2, where typing of function application is replaced by the 3 rules in figure 4. Theses rules each mimic one of the 3 reduction rules.

As before, we do not have the principal type property, but the problem is here limited to optional parameters. The partial inference algorithm we provide also checks for inconsistencies in the labeling: it is acceptable to apply a function whose parameters are labeled to non-labeled arguments, but applying a function without labels on some label, or applying with different labels both suggest an error. This checking can only be approximative, because of the above congruence on types, but it is again accurate enough on known functions. Any program typable in the standard system, and which does not use out-of-order application, is also typable in classic mode, meaning that the two modes have a significant intersection.

While call-by-value semantics must be slightly adapted for these new reduction rules, most of the discussion on values and compilation does still apply, since commutation is available between optional arguments.

8 Conclusion

We gave an almost complete account of how labeled and optional arguments were introduced in Objective Caml, from the dynamic semantics to the typed theory, and finally compilation. Notice that here types play the role they had in early times of programming languages: they provide information about the static shape of values, so as to be able to compile them efficiently. Both their restrictiveness, and some unspecified parts of the semantics, are deliberate: they intend to give way to an efficient, yet correct implementation.

We only presented a very succinct account of the classic mode semantics, since it is essentially a cut-down version of the full semantics, and does not present the same technical interest.

What we left out here is a detailed account of the compilation method, and a statement of its correctness. This is mostly straightforward, but an explicit framework is needed to prove the properties we just hinted. From a technical point of view, the more advanced compilation method proposed in [FG95] may in fact be more interesting, particularly if one wants to deal with a pure functional language.

Aknowledgements

This is an extension of previous work with Jun Furuse. Some modifications of the system are the result of discussions with Pierre Weis, Damien Doligez and Xavier Leroy. Anonymous referee comments were very helpful in improving both the form and the focus of this paper. This research is supported by a young researcher grant from the Ministry of Education and Science.

References

[AKG95] Hassan Aït-Kaci and Jacques Garrigue. Label-selective λ-calculus: Syntax and confluence. *Theoretical Computer Science*, 151:353–383, 1995.

[Dam98] Laurent Dami. A lambda-calculus for dynamic binding. *Theoretical Computer Science*, 192:201–231, 1998.

[FG95] 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.

[GAK94] Jacques Garrigue and Hassan Aït-Kaci. The typed polymorphic label-selective λ-calculus. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 35–47, 1994.

[Gar99] Jacques Garrigue. The Objective Label Trilogy 2.04. Available at `http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/`, 1999.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Mass., 1983.

[Lam88] John Lamping. A unified system of parameterization for programming languages. In *Proc. ACM Conference on LISP and Functional Programming*, pages 316–326, 1988.

[LDG+00] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system release 3.00, Documentation and user's manual*. Projet Cristal, INRIA, April 2000.

[Led81] Henry Ledgard. *ADA : An Introduction, Ada Reference Manual (July 1980)*. Springer-Verlag, 1981.

[Ste84] Guy L. Steele. *Common LISP : The Language*. Digital Press, 1984.