# A Label-Selective Lambda-Calculus with Optional Arguments and its Compilation Method

Jun P. Furuse and Jacques Garrigue

Research Institute for Mathematical Sciences, Kyoto University
Kitashirakawa-Oiwakecho, Sakyo-ku, Kyoto 606-01, JAPAN
E-mail: {furuse,garrigue}@kurims.kyoto-u.ac.jp

October 24, 1995

**Abstract**

The use of labels for argument passing has proved to be a useful extension to programming languages, particularly when we introduce optional arguments.

We propose here an adaptation of such a system, as can be found in Common LISP, to strongly-typed curried functional languages. For this we extend ML-style polymorphic typing to labeled arguments, out-of-order application, and optional arguments. We also provide a compilation method towards polymorphic lambda-calculus, which is proved syntactically sound for free reduction, and semantically correct for call-by-name semantics.

We implemented it, and showed the overhead caused by using this extension to be negligible when out-of-order parameter passing is used, and null when it is not.

**Topics:** language design, compilation methods, type theory.

## 1 Introduction

Labeled arguments are a way to both make more explicit the role of parameters in a function, and to permit out-of-order application. For instance, when we write the exponential function, `let exp b n` $= \mathtt{b}^{\mathtt{n}}$ in a Caml-like syntax[1], there is no "natural" order for the arguments. We would rather write it,

`let exp n` **base**`:b` $= \mathtt{b}^{\mathtt{n}}$

where we attach a label **base**: to `b`. This also means that we can now write `exp` **base**`:2 3` as well as `exp 3` **base**`:2` to compute $2^3$.

[1] *cf.* [16]. `let` denotes non-recursive definition, `fun x` $\rightarrow$ `e` $\lambda$-abstraction, `match e with` $\mathtt{p}_1 \rightarrow \mathtt{e}_1 \mid \ldots \mathtt{p}_n \rightarrow \mathtt{e}_n$ for matching, *etc...*

Extending this, default parameters permit to write functions in a compacter and clearer way. In the above example —and in a computer science context— $b$ will often happen to be 2. We would rather abbreviate it in such case. This can be done with the following definition.

$$\texttt{let exp n } \textbf{base}\texttt{:b}\langle 2\rangle = \texttt{b}^{\texttt{n}}$$

Now we can have two application patterns: `exp n`, which computes $2^n$, or `exp n` **base**`:b` (or reverse order), which computes $b^n$ as before.

But rather than default parameters, we chose to implement *optional* ones. We have a new type of matching pattern, $?l{:}p$ for optional matching, and two constructors, `Some` and `None`, respectively of arity 1 and 0. We are able to write a more optimal version of `exp`,

$$\begin{array}{l}\texttt{let exp n } ?\textbf{base}\texttt{:o} = \texttt{match o with } \texttt{Some b} \rightarrow \texttt{b}^{\texttt{n}} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid\ \texttt{None} \rightarrow \texttt{shiftl(1,n)}\end{array}$$

where shiftl may be a bitwise operation. Application patterns are still `exp n` and `exp n` **base**`:b`, but now we signal explicitly the presence or absence of a parameter to the called function by automatically adding a constructor.

The use of labeled and default/optional arguments has become a common feature in untyped programming languages, starting with Common LISP [14], and continuing with object-oriented languages (Smalltalk [6], the Tcl interface language [11], *etc...*) Among typed languages, ADA [8] already has this feature, but then typing is explicit, and currying is not allowed, which makes the task easy. Another attempt at typing was proposed by Dzeng and Haynes in [4], Common LISP. However, here again, the absence of currying reduces the problem to record typing, as solved by Rémy [12] among others; the question of compilation is not studied, but might be solved by the method suggested by Ohori for records with optional arguments [10].

Our basis for this extension is the label-selective lambda-calculus [1], a conservative extension of lambda-calculus with labeled application and abstraction. A first polymorphic typing system was proposed in [5]. In both cases argument order was completely abstracted.

In this paper we propose a new typing system, where we can distinguish between different abstraction orders, and handle optional arguments as "typeful syntactic sugar" (syntactic sugar depending on the typing of expressions); and give type inference and compilation algorithms for an ML-like language with labels and optionals. The choice of classical $\lambda$-calculus as target language was done regarding both the clarity of the translation induced, the optimization techniques available to compile the resulting $\lambda$-expression, and the easy integration into existing functional programming languages.

The structure of this paper is as follows. In the next section we define a simple calculus handling labels, the weak selective $\lambda$-calculus, and a type system, along with examples of how one can use labels. Section 3 describes a more complete ML-like language, relates it explicitly to the typing system, and provides a type reconstruction algorithm, which also produces a type-annotated weak selective $\lambda$-term corresponding to its input. In section 4, this type-annotated term is compiled into classical $\lambda$-calculus. Finally section 5 explains how compilation can be made more efficient. Argument of proofs are given in appendix.

# 2 Programming with labels

In this section, we first present types, and a simple $\lambda$-calculus, giving semantics to labeled arguments. We then give examples of the different possible uses of labeled arguments.

## 2.1 Types

Types we will use throughout this paper are the same as in Damas-Milner polymorphism [2], but for the labels in function types.

$$\begin{array}{lcll} \tau & ::= & b \mid \alpha \mid \tau \times \ldots \times \tau \mid \tau \text{ option} \mid l{:}\tau \to \tau \mid ?l{:}\tau \to \tau & \text{monotypes} \\ \sigma & ::= & \tau \mid \forall \alpha.\sigma & \text{polytypes} \end{array}$$

**option** is the sum type defined as `type` $\alpha$ `option` = `Some of` $\alpha$ | `None`. Optional labels $?l$: should only appear at the first level of types; arguments on optional labels may be omitted in applications.

We have a default label, 1:, which will be omitted in both types and terms.

## 2.2 A simple selective $\lambda$-calculus

The calculus we define here does not handle optional and default arguments. Those are only given semantics by translation into usual labeled arguments, during type reconstruction of the term (*cf.* Section 3, Figures 1 and 4), because their meaning is based on types[2].

This calculus differs from the selective $\lambda$-calculus of [5] by the absence of commutation between $\lambda$-abstractions[3]. This is why we call it *weak*.

Terms only differ from classical $\lambda$-calculus by the introduction of labels in abstraction and application. As usual, they are identified modulo $\alpha$-conversion.

$$M ::= x \mid c \mid \lambda l{:}x.M \mid M\ l{:}M$$

The new formulation of $\beta$-reduction is:

$$((\lambda l{:}M)\ l_1{:}N_1\ \ldots\ l_n{:}N_n\ l{:}N) \to (M[N/x]\ l_1{:}N_1\ \ldots\ l_n{:}N_n) \quad \text{if } (\forall i \leq n) l_i \neq l$$

**Theorem 1** *The* weak selective $\lambda$-calculus *is confluent.*

**Example 1** A trivial reduction.

$$(\lambda p{:}x.\lambda q{:}y.(x,y))\ q{:}a\ p{:}b \to (\lambda q{:}y.(b,y))\ q{:}a \to (b,a)$$

## 2.3 Labeled arguments

In the rest of this section we use Caml-like syntax for the above calculus.

Let's first see a few functions using only labeled parameters. Continuing with our first `exp` example, its type is,

```
let exp base:b n = b^n
exp = ⟨fun⟩ : base:int → int → int
```

[2] In this respect we differ from Common Lisp, where they are given reduction semantics.

[3] The calculus we propose here, and all the following systems, omit numeric labels. Their integration presents no difficulty, but would reduce the readability of this paper.

The above type means that you can apply `exp` to an integer, both on labels **base**: and 1:.

`exp` **base**:2 = ⟨fun⟩ : `int` → `int`
`exp` 3 = ⟨fun⟩ : **base**:`int` → `int`
`exp` 3 **base**:2 = 8 : `int`

Why would one want to use labels? We can see three kinds of reasons:

- to permit out-of-order partial applications
- not to have to remember parameter order
- to make programs more readable.

The first reason is clear. For a good understanding of the others we must study the role of function arguments. A first remark is that there is no "natural" argument order. For instance in Caml Light [9] the list folding function is defined as `it_list` : $(\alpha \to \beta \to \alpha) \to \alpha \to \beta$ `list` $\to \alpha$, but most people we discussed with have regularly to verify this order by looking at the type. One way to analyze arguments is to consider that, for most functions, like in object-oriented languages, we can distinguish an argument which is the object of the function, others being simply parameters for what does this function. Since `it_list` is a list manipulation function, considering its last argument as its object seems natural. A quick look at the library of Caml Light shows that in most cases the object appears to be the last argument of the function, enabling partial application on parameters. But this is not always the case: in the string and CamlTk modules, the object is the first argument. This last order is more natural from an object-oriented point of view, the function and its object becoming closer. For other parameters, we can see no rationale to prefer an order rather than another: this is often purely random.

What we suggest is a coherent labeling of functions. We do not need to label the object since it is unique (or at most two, for arithmetic functions for instance), but should label all parameters with their role or nature. Here are a few examples taken from the standard library. We didn't change parameter order, only added labels.

`map` : **fun**:$(\alpha \to \alpha) \to \alpha$ `list` $\to \alpha$
`it_list` : **fun**:(**st**:$\alpha \to \beta \to \alpha) \to$ **st**:$\alpha \to \beta$ `list` $\to \alpha$
`list_it` : **fun**:$(\alpha \to$ **st**:$\beta \to \beta) \to \alpha$ `list` $\to$ **st**:$\beta \to \beta$
`output_string` : **chan**:`out_channel` → `string` → `unit`
`blit_string` : `string` → **off**:`int` → **to**:`string` → **to_off**:`int` → **len**:`int` → `unit`

In these examples, **fun**: or **chan**: express the nature of their parameters, **st**: (for state), **off**: (for offset) or **to**: express their roles.

You can almost tell what a function does from its type. If I tell you once that in `blit_string` a substring of length **len** is copied from the object string, starting at offset **off**, to the **to** string, starting at offset **to_off**, wouldn't it be easier to remember than without labels? This is for our second reason.

The readability argument comes from our remark on argument order. There is no "natural" order in general, but there may be a more readable one when the function is used. It seems more natural to put the object of the function in first position. This also includes layout problems, like the length of each argument.

```
let rec quicksort order:cmp = fun [] → []
  | [a] → [a]
  | (a::l) → let (l1,l2) = it_list l st:([],[])
                  fun:(fun x st:(l1,l2) →
                          if cmp x a then (x::l1,l2) else (l1,x::l2))
             in (quicksort l1 order:cmp) @ (a::quicksort l2 order:cmp)
quicksort : order:(α → α → bool) → α list → α list
```

With labels we can pass `it_list`'s function as last argument, avoiding enlarging the distance between the function and its object, and without any auxiliary definition[4]. The label **fun**: at the head of the line makes clear that this is a parameter.

In [3] Di Cosmo proposes to use type isomorphisms in order to retrieve functions in libraries, according to their type. Labels can add flexibility and accuracy to this method.

## 2.4 Optional parameters

Optional parameters are in fact only a "typeful syntactic sugar", playing between $?l{:}\tau \to \tau'$ and $l{:}\tau$ `option` $\to \tau'$:

`fun` $?l{:}$`x` $\to$ `x` $= \langle \text{fun} \rangle : ?l{:}\alpha \to \alpha$ `option`

The example of introduction gets typed,

```
let exp n ?base:o = match o with  Some b → b^n
                                | None → shiftl(1,n)
exp = ⟨fun⟩ : int → ?base:int → int
```

The verification for presence or absence of a parameter is done at type checking of applications.

```
exp 3 = 8 : int
exp base:3 = ⟨fun⟩ : int → int
exp 3 base:3 = 27 : int
(fun x → x) exp = ⟨fun⟩ : int → int
```

In the first case type should be ?**base**:`int` → `int`. But all ?-labels appearing at the head of a type are then discarded, and `None` is passed as argument for each of them. As a result, it is interpreted as `exp 3` **base**:`None`, which evaluates to 8.

On the other hand, for the second case, **base:** is available. It is converted to `exp` **base**:(`Some 3`), and waits for its next parameter. Since the application typing rule is designed for simultaneous application on multiple arguments, this is also true for the third case, which evaluates to 27.

In the last example, when a function with optional parameters is passed as argument to another function, all its optional parameters are discarded, independently of their order (we actually pass `exp` **base**:`None`). Leaving them as optionals would result in incompleteness of the type inference.

Since the essential role of labels is in providing an interface for functions, we give the type og the creation method for a message widget, taken from our modified version of the CamlTk interface for Tcl/Tk.

[4]The usual way to write quicksort is to use an auxiliary definition, but the proliferation of those is not always good. In particular it makes it difficult to distinguish between which definition is local and which is not.

```
message__create : Widget → unit →
      ?anchor:Anchor → ?aspect:int → ?background:Color → ?foreground:Color →
      ?borderwidth:Units → ?cursor:Cursor → ?font:string → ?kanjifont:string →
      ?highlightbackground:Color → ?highlightcolor:Color →
      ?highlightthickness:Units → ?justify:Justification →
      ?name:string → ?padx:Units → ?pady:Units → ?relief:Relief →
      ?takefocus:bool → ?text:string → ?textvariable:TextVariable →
      ?width:Units → Widget
```

In the original version [13], options are represented as a list. Out of the visual inconvenient of a syntax departing largely from Tcl's, the problem of typing is unsolved. There is a unique sum type for all widget options, but all widgets are not able to handle the same options. As a result, when an illegal option is used a run-time error occurs. The only way to avoid these is to refer to the documentation. With optional arguments, the type of the creation function contains only the options handled by the message widget, enabling compile-time checking, and also permitting to verify accepted options by simply looking at the type[5].

Such optional parameters are necessary when one wants to interface a strongly typed language with any object-oriented one. But they are also a useful programming style when defining a library in the language itself, since they avoid defining many functions with various parameters. Why not define `assoc` : $\alpha \to$ **in**:$(\alpha \times \beta)$ `list` $\to$ ?**eq**:$(\alpha \to \alpha \to$ `bool`$) \to \beta$, allowing the user to give his own equality predicate when needed, and using the builtin one otherwise?

**Default parameters** They may be included as syntactic sugar:

$$\texttt{let exp n } \textbf{base}\texttt{:b}\langle 2\rangle = \texttt{b}^{\texttt{n}} \longrightarrow \begin{array}{l}\texttt{let exp n ?}\textbf{base}\texttt{:o} = \\ \quad \texttt{let b} = (\texttt{match o with Some b} \to \texttt{b} \mid \texttt{None} \to 2) \texttt{ in b}^{\texttt{n}}\end{array}$$

As such we consider them only as a subclass of optional parameters, eventually needing an optimized compilation method.

# 3 ML-like interface language and type reconstruction

In this section, we introduce an ML-like interface language with typeful syntactic sugar for optionals. We give both type inference rules and a type reconstruction algorithm which also generates type-annotated terms used for compilation.

## 3.1 Syntax

We extend terms of the simple calculus. Application now takes several arguments.

| | | | |
|---|---|---|---|
| $M$ | ::= | $x \mid c \mid (M, \ldots, M)$ | variable, constant, product |
| | $\mid$ | $\lambda l{:}x.M$ | lambda abstraction |
| | $\mid$ | $\lambda ?l{:}x.M$ | optional abstraction |
| | $\mid$ | $(M\ l{:}M\ \ldots\ l{:}M)$ | multi-application |
| | $\mid$ | let $x = M$ in $M$ | let expression |
| | $\mid$ | let rec $\{x_i = M_i\}_{i=1}^{n}$ in $M$ | recursive let |

[5]The `unit` type before optional parameters is there to permit partial application of `message__create` on a widget, without discarding options. A complete application will be written `create w name:"My name" ... ()`.

Types were defined in the previous section. $FTV(\sigma)$ is the set of free type variables of $\sigma$, using the standard definition. We can abbreviate function types using the following notation:

$$\{l_i{:}\tau_i\}_1^n \to \tau = l_1{:}\tau_1 \to \ldots \to l_n{:}\tau_n \to \tau$$

We define *similar types*, which differ only by some permutation on the order of labels in a monotype, and an *instantiation order* on types. For both relations we consider the reflexive and transitive closure.

$$\frac{l_1 \neq l_2}{l_1{:}\tau_1 \to l_2{:}\tau_2 \to \tau \simeq l_2{:}\tau_2 \to l_1{:}\tau_1 \to \tau} \qquad \frac{\tau_i \simeq \tau_i' \ (1 \leq i \leq n)}{\tau_1 \times \ldots \times \tau_n \simeq \tau_1' \times \ldots \times \tau_n'}$$

$$\frac{\tau_i \simeq \tau_i' \ \ (i \in \{1,2\})}{l{:}\tau_1 \to \tau_2 \simeq l{:}\tau_1' \to \tau_2'} \qquad \frac{\tau \simeq \tau'}{\tau \text{ option} \simeq \tau' \text{ option}} \qquad \forall\alpha.\sigma > \sigma[\tau/\alpha] \qquad \frac{\sigma > \sigma'}{\forall\alpha.\sigma > \forall\alpha.\sigma'}$$

The pattern $\rho$ matches all types except function types. $A_l(\tau)$ is the type of the first argument with label $l$ or $?l$ in $\tau$, and $R_l(\tau)$ is $\tau$ where this argument is omitted (*i.e.* $[?]l{:}A_l(\tau) \to R_l(\tau) \simeq \tau$). $L(\tau)$ indicates the multi-set of the labels occurring in $\tau$. If $\tau = \{l_i{:}\tau_i\}_1^n \to \rho$ , then $L(\tau)$ is the multi-set $\{l_i\}_1^n$ (identifying $?l$ with $l$). For example, $L(R_l(\tau)) = L(\tau) \setminus \{l\}$ (removing only one occurrence of $l$).

## 3.2 Inference rules

The inference rules for the interface language are given in Figure 1. The notation $\Gamma \vdash M \simeq \tau$ means $^{\exists}\tau' \simeq \tau, \Gamma \vdash M : \tau'$. *Typeof* contains the the constructors for the **option** type, and the following constant types (*cf.* 4.2.1 for associated $\delta$-rules):

$$\begin{array}{l} \mathit{case} : \forall\alpha.\forall\beta.(\alpha \text{ option} \times (\alpha \to \beta) \times \beta) \to \beta \\ \pi_i^n : \forall\alpha_1 \ldots \forall\alpha_n.(\alpha_1 \times \ldots \times \alpha_n) \to \alpha_i \\ \mathit{fix} : \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha \end{array}$$

We explain Abstraction, Optional abs, Application, and Letrec rules. The others are straightforward.

In the two abstraction rules, the abstracted variable is restricted to have a type in coerced form, that is, without optional labels $?l$. This guarantees the principality of the system. The coerce function $\lceil\cdot\rceil$ returns a type where every optional argument $?l{:}\tau$ is replaced by an argument $l{:}\tau$ **option**.

In the Optional abs rule, if an optional abstraction gets an argument $M$ typed $\tau$, it passes **Some** $M$ to the body, **None** otherwise. So the abstracted variable must have type $\tau$ **option**.

The Application rule takes a function and its arguments, and returns the applied term. It iteratively checks the types of the arguments and generates a resulting type. Furthermore, we erase the optional types which appear at the head of the obtained type. This does for the hidden **None** arguments.

The types of function domains are not checked with the types of the corresponding arguments themselves, but with their reduced types $\lfloor\tau\rfloor$. The reduce function $\lfloor\tau\rfloor$ erases all the optional arguments $?l{:}\theta$ from $\tau$. That is to avoid indeterminism of unification with optional arguments, which is discussed with the unification algorithm.

In Letrec rule, the "spine" condition is added to the usual rule. $\mathit{spine}(M_i)$ returns a template function type which contains information about the order of lambda and optional

Variable $\quad \Gamma\{x:\sigma\} \vdash x:\tau \;\; \sigma > \tau$ $\qquad$ Constant $\quad \Gamma \vdash c:\tau \;\; \text{Typeof}(c) > \tau$

Abstraction $\quad \dfrac{\Gamma\{x:\lceil\tau\rceil\} \vdash M:\tau'}{\Gamma \vdash \lambda l{:}x.M : l{:}\lceil\tau\rceil \to \tau'}$ $\qquad$ Optional abs. $\quad \dfrac{\Gamma\{x:\lceil\tau\rceil \text{ option}\} \vdash M:\tau'}{\Gamma \vdash \lambda ?l{:}x.M : ?l{:}\lceil\tau\rceil \to \tau'}$

Product $\quad \dfrac{\Gamma \vdash M_i:\tau_i \quad \text{for } 1 \le i \le n}{\Gamma \vdash (M_1,\ldots,M_n) : \tau_1 \times \ldots \times \tau_n}$

Application $\quad \dfrac{\Gamma \vdash M:\tau_0 \quad \Gamma \vdash N_i:\theta_i}{\Gamma \vdash (M\ l_1{:}N_1 \ldots l_m{:}N_m) : erase(\tau_m)}$ for $1 \le i \le m$ $\begin{cases} A_{l_i}(\tau_{i-1}) \simeq \lfloor\theta_i\rfloor \\ \tau_i = R_{l_i}(\tau_{i-1}) \end{cases}$

Let $\quad \dfrac{\Gamma \vdash M:\theta \quad \Gamma\{x:gen(\Gamma,\theta)\} \vdash N:\tau}{\Gamma \vdash \text{let } x = M \text{ in } N:\tau}$

Letrec $\quad \dfrac{spine(M_i) > \tau_i \quad \Gamma\{x_i:\tau_i\}_1^m \vdash M_i \simeq \tau_i \quad \Gamma\{x_i:gen(\Gamma,\tau_i)\}_1^m \vdash N:\tau}{\text{let rec } \{x_i = M_i\}_{i=1}^n \text{ in } N:\tau}$

Figure 1: Type inference rules

$gen(\Gamma,\tau) = \forall(FV(\tau) \setminus FV(\Gamma)).\tau$

$strip(\forall\alpha.\sigma) = strip([\alpha_{fresh}/\alpha]\sigma)$
$strip(\tau) = \tau$

$\lceil l{:}\tau_1 \to \tau_2\rceil = l{:}\tau_1 \to \lceil\tau_2\rceil$
$\lceil ?l{:}\tau_1 \to \tau_2\rceil = l{:}\tau_1 \text{ option} \to \lceil\tau_2\rceil$
$\lceil\tau_1 \times \ldots \times \tau_n\rceil = \lceil\tau_1\rceil \times \ldots \times \lceil\tau_n\rceil$
$\lceil\tau\rceil = \tau$ otherwise

$spine(c) = strip(\text{Typeof}(c))$
$spine(\lambda l{:}x.M) = l{:}\alpha \to spine(M)$
$spine(\lambda ?l{:}x.M) = ?l{:}\alpha \to spine(M)$
$spine(\text{let } x = M \text{ in } N) = spine(N)$
$spine(\text{let rec } \{x_i = M_i\}_{i=1}^n \text{ in } N) = spine(N)$
$spine(M) = \alpha$ otherwise
Here, all $\alpha$'s are fresh.

$erase(l{:}\tau_1 \to \tau_2) = l{:}\tau_1 \to \tau_2$
$erase(?l{:}\tau_1 \to \tau_2) = erase(\tau_2)$
$erase(\tau) = \tau$ otherwise

$\lfloor l{:}\tau_1 \to \tau_2\rfloor = l{:}\tau_1 \to \lfloor\tau_2\rfloor$
$\lfloor ?l{:}\tau_1 \to \tau_2\rfloor = \lfloor\tau_2\rfloor$
$\lfloor\tau_1 \times \ldots \times \tau_n\rfloor = \lceil\tau_1\rceil \times \ldots \times \lceil\tau_n\rceil$
$\lfloor\tau\rfloor = \tau$ otherwise

$sort(b) = base_b$
$sort(\tau_1 \times \ldots \times \tau_n) = prod_n$
$sort(l{:}\tau \to \tau) = fun$
$sort(\tau \text{ option}) = option$
$sort(\alpha)$ is left undefined

$Napp(M_{:\tau_1})_{:\tau_2} = (M_{:\tau_1}\ l_1{:}\text{None}\ \ldots\ l_n{:}\text{None})_{:\tau_2}$
where $\{l_1,\ldots,l_n\} = L(\tau_1) \setminus L(\tau_2)$

Figure 2: Some auxiliary functions for type inference rules

abstractions of $M_i$. We restrict $x_i$ to only have a type obtained by instantiation of that template. Without this constraint, we could infer the type $f : a{:}\alpha \to b{:}\beta \to \gamma$, for let rec $f = \lambda b{:}y.\lambda a{:}x.f\ a{:}x\ b{:}y$ in $\ldots$, which is due to the inference of $f\ a{:}x\ b{:}y$ but does not reflect the actual order of abstraction.

## 3.3 Labeled type unification

We give the unification algorithm for monotypes without optionals in Figure 3; we will use it for type reconstruction. The algorithm is given as rewriting rules for a system of

Failure $\dfrac{\varphi,\ \tau \simeq \tau'}{\bot}\ sort(\tau) \neq sort(\tau')$

$$\frac{\varphi,\quad \{l_i{:}\tau_i\}_1^n \to \tau \simeq \{l'_i{:}\tau'_i\}_1^m \to \tau}{\bot}\quad \{l_i\}_{i=1}^n \cap \{l'_i\}_{i=1}^m = \emptyset$$

Recurrence $\dfrac{\varphi,\ \alpha \simeq \tau \quad \alpha \in FV(\tau)}{\bot}$ Elimination $\dfrac{\varphi,\ \alpha \simeq \tau \quad \alpha \in Var(\varphi) \setminus Var(\tau)}{([\tau/\alpha]\varphi),\ \alpha \simeq \tau}$

Orientation $\dfrac{\varphi,\ \tau \simeq \alpha}{\varphi,\ \alpha \simeq \tau}$ Product $\dfrac{\varphi,\ \tau_1 \times \ldots \times \tau_n \simeq \tau'_1 \times \ldots \times \tau'_n}{\varphi,\ \tau_1 \simeq \tau'_1,\ \ldots,\ \tau_n \simeq \tau'_n}$

Redundancy $\dfrac{\varphi,\ \tau \simeq \tau}{\varphi}$ Option $\dfrac{\varphi,\ \tau_1\ \text{option} \simeq \tau_2\ \text{option}}{\varphi,\ \tau_1 \simeq \tau_2}$

$$\text{Decomposition}\quad \frac{\varphi,\quad \tau \simeq \tau'}{\varphi,\quad A_l(\tau) \simeq A_l(\tau'),\quad R_l(\tau) \simeq R_l(\tau')}\quad l \in L(\tau) \text{ and } l \in L(\tau')$$

$$\text{Completion}\quad \frac{\varphi,\quad \{l_i{:}\tau_i\}_1^n \to \rho \simeq \{l'_i{:}\tau'_i\}_1^m \to \rho'}{\varphi,\quad \rho \simeq \{l'_i{:}\tau'_i\}_1^m \to \alpha,\quad \rho' \simeq \{l_i{:}\tau_i\}_1^n \to \alpha}\quad \begin{array}{l}\{l_i\}_{i=1}^n \cap \{l'_i\}_{i=1}^m = \emptyset \\ \rho \neq \rho'\end{array}$$

Figure 3: Equation rewriting rules for unification

equations between similar types.

We do not allow to unify types with optional domains. They would introduce indeterminism. For instance, if we try to unify $?l{:}\texttt{int} \to \texttt{int}$ and $\alpha$, we cannot tell whether $\alpha$ should be unified with `int` or $?l{:}\texttt{int} \to \texttt{int}$, this only depends on the programmer's intention, and we do not know it.

A set of type equations $\varphi$ is in *solved form* if every equation has form $\alpha = \tau$ and the type variable $\alpha$ occurs nowhere else in $\varphi$. Such a solved-form defines a variable substitution that can be applied to type expressions.

Upon termination, having started from a set $\varphi$ of equations, the resulting set is either $\bot$, the inconsistent equation indicating that no solution exists, or $\text{sol}(\varphi)$, a set of equations in solved form equivalent to $\varphi$.

**Theorem 2** *There is a terminating algorithm to solve the mgu of a set of type equations.*

The solution obtained with this rewriting system is unique modulo type similar equivalence of right hand side types. For example, if $l \neq r$, then $\alpha \simeq l{:}\tau_l \to r{:}\tau_r \to \rho,\quad \alpha \simeq r{:}\tau_r \to l{:}\tau_l \to \rho$ has two solutions, each of them consisting of one of these equations. But they are type similar. Which solution is selected for type reconstruction depends on how the unification algorithm chooses equations to solve. The term $\lambda f.(f\ a{:}1\ b{:}2,\ f\ b{:}3\ a{:}4)$ can be typed either $(a{:}int \to b{:}int \to \alpha) \to \alpha$ or $(b{:}int \to a{:}int \to \alpha) \to \alpha$. Since they are type similar, they can be used in the same contexts.

## 3.4 Type reconstruction algorithm

We define a type reconstruction algorithm in Figure 4. All $\alpha$'s are fresh. $sol(\varphi)$ is the mgu of $\varphi$ by the unification algorithm.

$$
\begin{array}{lcll}
\mathcal{T}_{\mathcal{P}}(\Gamma,\ c) & = & \langle \emptyset,\ \tau,\ c_{:\tau}\rangle & \text{where } \tau = \mathit{strip}(\mathrm{Typeof}(c)) \\
\mathcal{T}_{\mathcal{P}}(\Gamma,\ x) & = & \langle \emptyset,\ \tau,\ x_{:\tau}\rangle & \text{where } \tau = \mathit{strip}(\Gamma(x)) \\
\mathcal{T}_{\mathcal{P}}(\Gamma,\ (M_1,\ldots,M_n)) & = & \langle \mathit{sol}(\bigcup_1^n \varphi_i),\ \tau_1\times\ldots\times\tau_n,\ (M'_1,\ldots,M'_n)_{:\tau_1\times\ldots\times\tau_n}\rangle & \\
\quad \text{where} & & \langle \varphi_i,\ \tau_i,\ M'_i\rangle = \mathcal{T}_{\mathcal{P}}(\Gamma,\ M_i) & \\
\mathcal{T}_{\mathcal{P}}(\Gamma,\ \lambda l{:}x.M) & = & \langle \varphi,\ l{:}\alpha\to\tau,\ (\lambda l{:}x.M')_{:l:\alpha\to\tau}\rangle & \\
\quad \text{where} & & \langle \varphi,\ \tau,\ M'\rangle = \mathcal{T}_{\mathcal{P}}(\Gamma[x:\alpha],\ M) & \\
\mathcal{T}_{\mathcal{P}}(\Gamma,\ \lambda ?l{:}x.M) & = & \langle \varphi,\ l{:}?\alpha\to\tau,\ (\lambda l{:}x.M')_{:l:?\alpha\to\tau}\rangle & \\
\quad \text{where} & & \langle \varphi,\ \tau,\ M'\rangle = \mathcal{T}_{\mathcal{P}}(\Gamma[x:\alpha\ \mathrm{option}],\ M) &
\end{array}
$$

$$
\mathcal{T}_{\mathcal{P}}(\Gamma,\ (M\ l_1{:}N_1\ldots l_n{:}N_n)) = \langle \psi_n,\ \tau, \mathit{Napp}((M'_{:\{l_i:\theta'_i\}_1^n\to\tau_n}\ l_1{:}(N''_1)_{:\theta'_1}\ \ldots\ l_n{:}(N''_n)_{:\theta'_n})_{:\tau_n})_{:\tau}\rangle
$$

$$
\text{where}\ \begin{cases}
\langle \varphi_0,\ \tau_0,\ M'\rangle = \mathcal{T}_{\mathcal{P}}(\Gamma,\ M) \\
\psi_i = \mathit{sol}(\bigcup_{j=0}^{i}\varphi_j\cup\varphi'_j) \\
\langle \varphi_i,\ \theta_i,\ N'_i\rangle = \mathcal{T}_{\mathcal{P}}(\psi_{i-1}(\Gamma),\ N_i) \\
\langle \theta'_i, N''_i\rangle = \begin{cases} \langle \lfloor\theta_i\rfloor\ \mathrm{option}, \mathrm{Some}(\mathit{Napp}(N'_i)_{:\lfloor\theta_i\rfloor})\rangle \\ \quad \text{if } ?l_i \text{ is the first occurrence of } l_i \text{ in } \tau_i \\ \langle \lfloor\theta_i\rfloor, \mathit{Napp}(N'_i)_{:\lfloor\theta_i\rfloor}\rangle \qquad \text{otherwise} \end{cases} \\
\varphi'_i = \mathit{sol}\{\lceil\tau_{i-1}\rceil \simeq l_i{:}\theta'_i\to\alpha\} \\
\tau_i = R_{l_i}(\psi_i(\tau_{i-1})) \\
\tau = \mathit{erase}(\tau_n)
\end{cases}
$$

$$
\begin{array}{lcl}
\mathcal{T}_{\mathcal{P}}(\Gamma,\ \mathrm{let}\ x = M\ \mathrm{in}\ N) & = & \langle \mathit{sol}(\varphi\cup\varphi'),\ \tau',\ (\mathrm{let}\ x_{:\sigma} = M'\ \mathrm{in}\ N')_{:\tau'}\rangle \\
\quad \text{where} & & \begin{cases} \langle \varphi,\ \tau,\ M'\rangle = \mathcal{T}_{\mathcal{P}}(\Gamma,\ M) \\ \sigma = \mathit{gen}(\varphi(\Gamma),\varphi(\tau)) \\ \langle \varphi',\ \tau',\ N'\rangle = \mathcal{T}_{\mathcal{P}}(\varphi(\Gamma)[x:\sigma],\ N) \end{cases}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{T}_{\mathcal{P}}(\Gamma,\ \mathrm{let\ rec}\ \{x_i = M_i\}_{i=1}^n\ \mathrm{in}\ N) = \langle \mathit{sol}(\varphi\cup\varphi'),\tau, \\
\qquad (\mathrm{let}\ x_{:\sigma} = \mathit{fix}(\lambda x.[\pi_i^n\ x/x_i]_{i=1}^n((M'_1)_{:\tau_1},\ldots,(M'_n)_{:\tau_n}))\ \mathrm{in}\ [\pi_i^n\ x/x_i]_{i=1}^n N')_{:\tau}\rangle
\end{array}
$$

$$
\text{where}\ \begin{cases}
\tau_i = \mathit{spine}(M_i) \\
\langle \varphi_i,\ \tau'_i,\ M'_i\rangle = \mathcal{T}_{\mathcal{P}}(\varphi'_k(\Gamma[x_i:\tau_i]_1^n),\ M_i) \\
\varphi'_k = \mathit{sol}(\bigcup_{i=1}^{k-1}\varphi_i) \\
\sigma_i = \mathit{gen}(\varphi(\Gamma),\varphi(\tau_i)) \\
\langle \varphi',\ \tau,\ N'\rangle = \mathcal{T}_{\mathcal{P}}(\varphi(\Gamma)[x_i:\sigma_i]_{i=1}^n,\ N)
\end{cases}
$$

Figure 4: Type reconstruction algorithm

$\mathcal{T}_{\mathcal{P}}$ takes a typing environment $\Gamma$ and a term $M$ of the interface language, and returns a tuple $\langle\varphi,\tau,M'\rangle$, where $\varphi$ is a set of type equations in solved form (that can be used as a type substitution). $\tau$ is such that $\mathit{gen}(\varphi(\Gamma),\varphi(\tau))$ is the principal type of $M$. $\mathcal{T}_{\mathcal{P}}$ also returns a type-annotated term $M'$ for $M$.

When we create a new environment by adding types of subterms (in application, let, and letrec part), we apply the substitutions obtained in the type reconstruction of subterms to normalize it. It is not needed in ordinary ML style type reconstruction. But we want the most accurate information about abstraction order to be in the newly created environment.

In let rec, each declaration is inferred under an extended environment in which the declared variables are assigned type templates generated by *spine*. Since these variables

have (optional) monotypes, the mgu is obtained by unification of their coerced forms.

A type-annotated term $M'$ is created by resolving all syntactic additions — optionals and letrec-expressions — from $M$. The well-typed term is $\varphi(\lceil M' \rceil)$, where $\lceil M' \rceil$ is a type-annotated term obtained by replacing each type annotation $\tau$ of $M'$ by $\lceil \tau \rceil$. Optional abstractions are replaced by normal lambda abstractions. In application, for each applied argument $N_i$, we use $Napp(N'_{i:\theta_i})_{:\lfloor \theta_i \rfloor}$, which is $N_i$ applied to **None** on all its optional labels. Furthermore, if the label $N_i$ is applied to is optional, we apply **Some** to it. For erased optional labels, **None** is applied. Finally, recursive let-expressions are rewritten with the $fix$ constant.

Here is the definition of type-annotated terms, based on the simple calculus of section 2.

**Definition 1 (type-annotated term)** *The type-annotated term $M_{:\tau}$ stands for $M$ of type $\tau$ (monotype). We introduce three new kinds of term, product, type coercion, and let definition.*

$$M_{:\tau} ::= x_{:\tau} \mid c_{:\tau} \mid (\lambda l{:}x.M_{:\tau})_{:\tau} \mid (M_{:\tau}\ l{:}M_{:\tau})_{:\tau} \mid (M_{:\tau}, \ldots, M_{:\tau})_{:\tau} \mid (M_{:\tau})_{:\tau} \mid \text{let } x_{:\sigma} = M_{:\tau} \text{ in } M_{:\tau}$$

*Well-typed terms under an environment $\Gamma$ are those inferred by the following rules:*

$$\frac{\Gamma(x) > \tau}{\Gamma \vdash x_{:\tau}} \qquad \frac{\text{Typeof}(c) > \tau}{\Gamma \vdash c_{:\tau}} \qquad \frac{\Gamma \vdash M^i_{:\tau_i} \text{ for } 1 \leq i \leq n}{\Gamma \vdash (M^1_{:\tau_1}, \ldots, M^n_{:\tau_n})_{:\tau_1 \times \ldots \times \tau_n}}$$

$$\frac{\Gamma[x \mapsto \theta] \vdash M_{:\tau}}{\Gamma \vdash (\lambda l{:}x.M_{:\tau})_{:l:\theta \to \tau}} \qquad \frac{\Gamma \vdash M_{:l:\theta \to \tau} \quad \Gamma \vdash N_{:\theta}}{\Gamma \vdash (M_{:l:\theta \to \tau}\ N_{:\theta})_{:\tau}}$$

$$\frac{\Gamma \vdash M_{:\tau} \quad \tau \simeq \tau'}{\Gamma \vdash (M_{:\tau})_{:\tau'}} \qquad \frac{\Gamma \vdash M_{:\tau} \quad \Gamma[x \mapsto \forall \alpha_1 \ldots \forall \alpha_n.\tau] \vdash N_{:\tau'}}{\Gamma \vdash \text{let } x_{:\forall \alpha_1 \ldots \forall \alpha_n.\tau} = M_{:\tau} \text{ in } N_{:\tau'}} \ (\forall i \leq n)\ \alpha_i \notin FTV(\Gamma)$$

**Theorem 3** *If a term $M$ of the interface language is typable, $\mathcal{T}_{\mathcal{P}}$ returns the principal type of $M$ and a well-typed annotated term for $M$. Otherwise $\mathcal{T}_{\mathcal{P}}$ reports a type error.*

# 4 Compilation

From any type-annotated weak selective $\lambda$-term, we can compile away labels, to obtain a classical $\lambda$-calculus term. We propose here a method to do that, and prove its syntactic soundness and semantic correctness.

## 4.1 Compilation method

$[M_{:\tau'}]_\tau$, the compilation of the type-annotated selective $\lambda$-term $M_{:\tau'}$ under the labeled type $\tau$" is defined as follows. $[M]_\tau$ is an abbreviation for $[M_{:\tau}]_\tau$.

$$\begin{array}{lcllcl}
[M_{:\tau'}]_\tau & = & \mathcal{T}([M]_{\tau'}, \tau', \tau) & [\lambda l{:}x.M_{:\tau'}]_{l:\theta \to \tau} & = & \lambda x.[M_{:\tau'}]_\tau \\
[x]_\tau & = & x & [M_{:\tau'}\ l{:}N_{:\theta}]_\tau & = & [M_{:\tau'}]_{l:\theta \to \tau}\ [N]_\theta \\
[c]_\tau & = & c & [\text{let } x_{:\sigma} = M_{:\tau} \text{ in } N_{:\tau'}]_{\tau'} & = & \text{let } x = [M]_\tau \text{ in } [N]_{:\tau'}
\end{array}$$

$$
\begin{array}{lll}
\text{(let)} & \text{let } x_{:\forall\alpha_1\ldots\alpha_n.\tau} = M_{:\tau} \text{ in } N_{:\tau'} \to [M_{:\tau}/x_{:\forall\alpha_1\ldots\alpha_n.\tau}]N_{:\tau'} & \\
(\beta) & ((\ldots(\lambda l{:}x.M_{:\tau})_{:\tau_1}\ldots)_{:\tau_n}\ l_1{:}N^1_{:\theta_1}\ldots\ l_n{:}N^n_{:\theta_n}\ l{:}N_{:\theta})_{:\tau'} & \text{if } \forall i \le n, l_i \neq l \\
 & \to ((\ldots([N'/x]M_{:\tau})_{:\nu_1}\ldots)_{:\nu_n}\ l_1{:}N^1_{:\theta_1}\ldots\ l_n{:}N^n_{:\theta_n})_{:\tau'} & \\
 & \text{where } N' = (\ldots(N_{:\theta})_{:A_l(\tau_n)}\ldots)_{:A_l(\tau_1)} \text{ and } \nu_i = R_l(\tau_i) & \\
(\delta) & ((\ldots c_{:l:\theta_1\to\tau_1}\ldots)_{:l:\theta_n\to\tau_n}\ l{:}N_{:\theta_n})_{:\tau_n} \to (\ldots\delta(c, l{:}(\ldots N_{:\theta_n}\ldots)_{:\theta_1})_{:\tau_1}\ldots)_{:\tau_N} & \\
 & & \\
\text{Convert} & ((M_{:l:\theta\to\tau}\ l{:}N_{:\theta})_{:\tau})_{:\tau'} \to ((M_{:l:\theta\to\tau})_{:l:\theta\to\tau'}\ l{:}N_{:\theta})_{:\tau'} & \text{if } \tau_1 \simeq \tau_2 \\
\text{Simplify} & (M_{:\tau})_{:\tau} \to M_{:\tau} &
\end{array}
$$

Figure 5: Typed reduction rules

The above definition just handles type annotations. The real work is delegated to the type conversion $\mathcal{T}$. (In the second case, $\xi$ is a permutation of $[1, n]$.)

$$
\begin{array}{ll}
\mathcal{T}(M, \tau, \tau) & = M \\
\mathcal{T}(M, \{l_{\xi_i}{:}\tau_{\xi_i}\}_1^n \to \tau, \{l_i{:}\tau'_i\}_1^n \to \tau') & = \lambda^* x_1 \ldots x_n.\mathcal{T}(M\ \mathcal{T}(x_{\xi_1}, \tau'_1, \tau_{\xi_1}) \ldots \mathcal{T}(x_{\xi_n}, \tau'_n, \tau_{\xi_n}), \tau, \tau') \\
\mathcal{T}(M, \tau_1 \times \ldots \times \tau_n, \tau'_1 \times \ldots \times \tau'_n) & = (\lambda^+ x.(\mathcal{T}(\pi_1^{n*}\ x, \tau_1, \tau'_1), \ldots, \mathcal{T}(\pi_n^{n*}\ x, \tau_n, \tau'_n)))\ M \\
\mathcal{T}(M, \tau \text{ option}, \tau' \text{ option}) & = case^*(M, \lambda x.\text{Some}(\mathcal{T}(x, \tau, \tau')), \text{None})
\end{array}
$$

The marks ($*$ and $+$) introduced by type conversion do not change the meaning of terms. We can use them to optimize the translation: $\beta$ and $\eta$-redexes including a marked $\lambda^*$ can be reduced without changing intuitive semantics — for $\beta$-redexes, use of the abstracted variable is linear, and for $\eta$-redexes, they were absent before the translation. —This is also true for $\lambda^+$, but only when applied to an actual tuple, and the following $\pi^*$'s are immediately $\delta$-reduced. Similarly *case*$^*$'s can be $\delta$-reduced.

We note $[M]^*_\tau$ for the optimized compilation of $M$, that is $[M]_\tau$ where all the above was done. We define also a weaker compilation, $[M]^{*n}_\tau$, where all $\eta*$-redexes are reduced, but other $*$ and $+$ redexes are only reduced when they appear at the leftmost position of the compiled term (*i.e.* $(\lambda^* x.M)\ N_1 \ldots N_n$, $(\lambda^+ x.M)\ (N_{11}, \ldots, N_{1n}) \ldots N_n$, or $case^*(M, N, P)\ N_1 \ldots N_n$). That is, we only reduce at compilation what could be done by the call-by-name $\lambda$-calculus.

## 4.2 Correctness of compilation

Before proving correctness, we need some referent calculus before compilation. For this we add a typed reduction system to our type-annotated terms.

### 4.2.1 Typed reduction

The typed reduction rules are in Figure 5. When the variable to substitute is not annotated by a polytype, variable substitution needs no special care, out of renaming bound variables as usual. But when it is annotated by a polytype, as may happen with **let**-conversion, we need to do some type variable substitution on all annotations of the substituted term:

$$
[M_{:\tau}/x_{:\forall\alpha_1\ldots\alpha_n.\tau}]x_{:\tau'} = S(M_{:\tau}) \qquad \text{if } \begin{cases} D_S = \{\alpha_1, \ldots, \alpha_n\} \\ S(\tau) = \tau' \end{cases}
$$

In ($\beta$), there should be only first conversions, then applications on the path between $M$ and $N$. Type conversions are pushed inside by *Conversion*, or eliminated by *Simplify*.

($\delta$) applies only if there is a corresponding $\delta$-rule[6].

$$\begin{array}{lcl}
\delta(\pi_i^n, 1{:}(M_1, \ldots, M_n)) & = & M_i \\
\delta(\mathit{case}, 1{:}(\text{Some } M, N, P)) & = & N\ 1{:}M \\
\delta(\mathit{case}, 1{:}(\text{None}, N, P)) & = & P \\
\delta(\mathit{fix}, 1{:}M) & = & M\ 1{:}(\mathit{fix}\ 1{:}M)
\end{array}$$

All these rules preserving type information,

**Proposition 1 (subject reduction)** *For each typed reduction step $M_{:\tau} \to N_{:\tau'}$, if $\Gamma \vdash M_{:\tau}$ then $\tau = \tau'$ and $\Gamma \vdash N_{:\tau'}$.*

**Theorem 4** *The* type-annotated weak selective $\lambda$-calculus *is confluent.*

### 4.2.2 Correctness properties

One may think of several ways to prove a compilation algorithm correct. The most usual one is to use a common model before and after compilation, and prove the algorithm correct with respect to this model. We do this for call-by-name evaluation.

The property we prove first here is slightly different. Since we are translating into the lambda calculus, and one may think of various models (or evaluation strategies) for it, we do not commit to one of them, and prove a more general property on rewrite systems.

**Theorem 5 (syntactic soundness)** *If $M_{:\tau} \to N_{:\tau}$ in the type-annotated weak selective $\lambda$-calculus, then $[M]^*_\tau \xrightarrow{*} [N]^*_\tau$.*

**Theorem 6 (semantic correctness)** $[\cdot]^*_\tau$ *compilation is correct with respect to call-by-name operational semantics. That is, if $M_{:\tau}$ is a closed term of base type, then*

1. *when call-by-name strategy leads to a normal form $N_{:\tau}$, then $[M]^*_\tau$ is reduced to the normal form $[N]^*_\tau$ by the same strategy.*

2. *when call-by-name strategy leads to an infinite reduction, its application to $[N]^*_{:\tau}$ also leads to an infinite reduction.*

Remark that we do not have correctness with respect to call-by-value semantics: every time call-by-value evaluation of $M_{:\tau}$ terminates, call-by-value evaluation of $[M]^*_{:\tau}$ also terminates, which gives us soundness, but since we introduce abstractions in the compilation process, the inverse is not true.

[6]This is a simplified version: we require the parameter of the $\delta$-rule to be the first argument. This is not a limitation since we recover full generality by an $\eta$-expansion for the parameter.

# 5 Efficiency considerations

The goal of the above compilation method (including the simplifications done by type inference) is to produce code, of course correct, but also with a minimal overhead against a non-labeled program. In particular the choice to compile towards classical lambda-calculus rather than a new abstract machine avoids handicapping programs using labels in a non-essential way (with no out-of-order partial applications). Actually, in a program containing only full applications of functions (or in-order partial applications), the $*$-optimized compilation just reorders the arguments in function applications, to match the definition template.

We still need to extend a little $*$-compilation to handle efficiently out-of-order partial applications. By substituting all marked redexes at compilation, we may happen to move a function application inside an abstraction:

$$[M_{:p:\theta\to q:\tau\to\tau'}\ q{:}(f\ a)_{:\tau}]^*_{p:\theta\to\tau'} = (\lambda^* x.\lambda^* y.M\ y\ x)\ (f\ a) \to \lambda^* y.M\ y\ (f\ a)$$

This results potentially into multiple evaluations of $(f\ a)$. To avoid this, we: 1) put terms in *Convert/Simplify* normal form before compiling them; 2) introduce let-definitions to keep non-values (head-reducible terms, that includes applications and let-definitions containing applications) out of abstractions; 3) limit compile-time $\beta$-reduction to values.

$$\text{let}^*\ z = f\ a\ \text{in}\ (\lambda^* x.\lambda^* y.M\ y\ x)\ z \to \text{let}^*\ z = f\ a\ \text{in}\ \lambda^* y.M\ y\ z$$

These let-definitions are transparent: $(\text{let}^*\ x = N\ \text{in}\ M)\ N' \to \text{let}^*\ x = N\ \text{in}\ M\ N'$.

With this compilation method, we only depart from usual operational semantics (either lazy or strict) in that no reduction of the function is done when there is an "hole" in the partial application of a function, like in $(f_{:p:i\to q:i\to r:i\to i}\ p{:}a\ r{:}b)$ : $a$ and $b$ are shared, but $f\ a$ is not reduced. One may of course force the reduction by doing the application in two stages, but this problem is generally not very important: real evaluation of partial application is very rare, *a fortiori* in such an unintentional way.

As a conclusion, no overhead is involved by using labels, other than what would have been needed anyway in order to write an out-of-order partial application, and semantics are kept even with strict evaluation, except for side-effects in out-of-order partial applications (we should give a warning for functions that return an unprotected function type after having side-effects, they are ambiguous).

For optional parameters, two sources of inefficiency can be seen. One is the need for using an explicit constructor `Some` for all actually passed parameters, the other is that even if a parameter is not passed, `None` must be passed in its place. In most implementations, the first inefficiency can be reduced by a little cooperation from the back-end. This is based on the following remark: a value of type $(\ldots(\tau\ \texttt{option})\ldots)\ \texttt{option}$ can only be either $\texttt{Some}(\ldots\texttt{Some}(a_{:\tau})\ldots)$ or $\texttt{Some}^n(\texttt{None})$ (where $0 \le n \le$ the number of option levels in the type). In an implementation distinguishing between scalars and pointers by the highest bit of words (like Caml Light [9] for instance), we can view the latter as an array containing all constants of this form (a finite number is enough), its location completely distinct from any other value. As a result, we do not need the `Some` constructor in representations, and all option manipulation can be done without any heap access. On the other hand, the second problem seems intrinsic to currying.

# 6 Conclusion

Labeled and optional arguments were not handled by strongly typed functional programming languages. This is now done... in theory!

Our proposal leaves to the user the choice between writing labeled programs or not, without adding any new cost to the last alternative.

Advantages of labeled programs and libraries are

- Better readability of the source code.
- More information in the type.
- No parameter order to remember.
- Easier to add parameters to library functions.
- Syntax closer to object-oriented style.

Concern about semantics remains. While integration of this system in a lazy functional programming language like Haskell [7] creates no difficulty (in the worse —and exceptional— case, some sharing will be lost), for ML-style strict languages with side-effects we must be more careful. Faulty programs have in common a dangerous use of partial application, producing side-effects. If all functions are defined such that they may only have side-effects when all their parameters are applied, this concern disappears. Since such a constraint also makes possible other source code transformations, we think that it may be worth considering it.

# References

[1] Hassan Aït-Kaci and Jacques Garrigue. Label-selective $\lambda$-calculus: Syntax and confluence. In *Proc. of the Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 24–40, Bombay, India, 1993. Springer-Verlag LNCS 761. Journal version to appear in *Theoretical Computer Science*.

[2] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[3] Roberto Di Cosmo. *Isomorphisms of types: from $\lambda$-calculus to information retrieval and language design*. Progress in Theoretical Computer Science. Birkhauser, 1995. ISBN-0-8176-3763-X.

[4] Hsianlin Dzeng and Christopher T. Haynes. Type reconstruction for variable-arity procedures. In *Proc. ACM Conference on LISP and Functional Programming*, pages 239–249, 1994.

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

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

[7] Haskell committee. Haskell report 1.2. *SIGPLAN Notices*, 27(5), May 1992.

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

[9] Xavier Leroy. *The Caml Light System Documentation, release 0.7.* INRIA, Rocquencourt, France, July 1995. Available at URL http://pauillac.inria.fr/caml/man-caml.

[10] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, to appear.

[11] John K. Ousterhout. *Tcl and the Tk toolkit.* Addison-Wesley, 1994. ISBN 0-201-63337-x.

[12] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 77–87, 1989.

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

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

[15] Masako Takahashi. Parallel reductions in $\lambda$-calculus. *Information and Computation*, 118:120–127, 1995.

[16] Pierre Weis et al. *The CAML Reference Manual, version 2.6.1.* Projet Formel, INRIA-ENS, 1990.

# A Proofs

**Theorem 1** *The* weak selective $\lambda$-calculus *is confluent.*

PROOF: Like for classical $\lambda$-calculus, this is easily proved by parallel reductions [15]. □

**Theorem 2** *There is a terminating algorithm to solve the mgu of a set of type equations.*

PROOF: **System correctness.** In each rule, any unifier of the denominator unifies also the numerator, and conversely, any unifier of the numerator can be extended to a unifier of the denominator by introducing new variables missing in the numerator.

We only detail the case of Completion. If $\sigma$ is a solution of the numerator, then

$$\{l_i{:}\sigma(\tau_i)\}_1^n \to \sigma(\rho) \simeq \{l'_i{:}\sigma(\tau'_i)\}_1^m \to \sigma(\rho')$$

Since all $l_i$ and $l'_i$ are distinct each other, $\sigma(\rho)$ must have labels which correspond with $\{l'_i{:}\tau'_i\}_1^m$. More correctly,

$$\sigma(\rho) = \{l'_i{:}\tau'_i\}_1^m \to \tau \qquad \text{and} \qquad \sigma(\rho') = \{l_i{:}\tau_i\}_1^n \to \tau'$$

for some $\tau \simeq \tau'$. Then if we extend $\sigma$ such that $\sigma(\alpha) = \tau$, $\sigma$ can be a solution of the denominator. The converse is clear by construction.

**Termination.** We show that the algorithm terminates if we give priority to failure rules. A variable is solved when it appears only once and as the left-hand-side of an equation. And the size of a type is the total number of labels, variable occurrences, base types, $\times$ and

option it contains. We adopt (*number of unsolved variables, sum of sizes*) as lexicographical measure to show the termination.

All rules except Completion reduce the measure. For Completion, if one of $\rho$ and $\rho'$ is not a variable, then next step is a failure. Otherwise we introduce one new variable, but solve two.

**Well-formedness of the result.** Last we must show that our result is in solved form. First, in every equation, at least one side is a solved variable. If the two sides are functional types, then either Decomposition, Completion, or one of failure rules applies. If one side is a base type, then the other side is a solved variable, otherwise Redundancy or Failure applies. If the two sides are variables, then at least one is solved.

We construct the substitution $\sigma$ by taking for each equation $\alpha = \tau$, $\alpha$ solved, $\sigma(\alpha) = \tau$. $\sigma$ is a most general unifier of the final system, and as a consequence, if we suppress definitions for all variables introduced by Completion, $\sigma'$ is a most general unifier of the original system. □

**Theorem 3** *If a term $M$ of the interface language is typable, $\mathcal{T}_{\mathcal{P}}$ returns the principal type of $M$ and a well-typed annotated term for $M$. Otherwise $\mathcal{T}_{\mathcal{P}}$ reports a type error.*

PROOF: (Sketch) All type inference rules correspond exactly to one case of the type reconstruction algorithm, except generalization and instantiation. Since the algorithm only adds the constraints insured by type inference rules and introduces generalization and instantiation at most general places, it is sound and complete.

Together with the principal type $\tau$, $\mathcal{T}_{\mathcal{P}}$ returns a type-annotated term of type $\lceil\tau\rceil$. When $\mathcal{T}_{\mathcal{P}}$ does not introduce any polymorphism, it is easy to check $\mathcal{T}_{\mathcal{P}}$ returns a well-typed annotated term, particularly in application $M'$ is type converted to the order and types of its arguments.

When introducing polymorphism (let and let rec), type-annotated terms of each declarations are generalized, but these generalizations are all syntactically correct for the type annotated calculus. □

**Proposition 1 (subject reduction)** *For each typed reduction step $M_{:\tau} \rightarrow N_{\tau'}$, if $\Gamma \vdash \in M_{:\tau}$, then $\Gamma \vdash N_{:\tau'}$ and $\tau = \tau'$.*

PROOF: This is clear for *Convert* and *Simplify*.

For ($\beta$) and $\delta$-rule, we reconstruct all the type conversions necessary.

Thanks to type-variable substitution when needed, (let) works also. □

**Theorem 4** *The* type-annotated weak selective $\lambda$-calculus *is confluent.*

PROOF: (sketch). We first remark that *Convert* and *Simplify* commute with themselves and each other. This solves their critical pairs. This is also true for $\delta$-rules, except *fix*.

However one-step ($\beta$), (let) and $\delta$-*fix* may duplicate some redexes, which prohibits immediate commutation. Following Takahashi [15], we define their parallel reduction as the simultaneous reduction of any number of non-overlapping ($\beta$), (let) and $\delta$-*fix* redexes. The result for classical lambda-calculus directly extends to give us local confluence of parallel ($\beta$)-(let)-$\delta$-*fix*, and confluence of the combination of ($\beta$), (let) and $\delta$-*fix*.

We may then show that one-step ($\beta$), (let) or $\delta$-*fix* commutes with multiple steps of other rules. As a result multi-step ($\beta$)-(let)-$\delta$-*fix* commutes with multiple steps of other rules.

If we have two confluent rewriting systems, and they commute with each other, then their union is confluent. □

**Theorem 5 (syntactic soundness)** *If $M_{:\tau} \to N_{:\tau}$ in the type-annotated weak selective $\lambda$-calculus, then $[M]^*_\tau \xrightarrow{*} [N]^*_\tau$.*

PROOF: Remark first that *Convert* and *Simplify* change nothing to the compiled term: see the translation of type coercion when $\tau' \simeq \tau$, and that of application, and apply $*$-normalization afterwards.

For $(\beta)$ and (let), the proof is by induction on the structure of the term in which the variable is substituted. After reducing the translated redex in $[M]^*_\tau \to M'$, the term we obtain contains potentially reducible $*$ and $+$-redexes, created by the application of the substituted term to its new contexts, or by the reduction of a redex. By reducing them, we reach $[N]^*_\tau$. □

**Lemma 1** *If $\tau$ is a base type, and $M_{:\tau} \to N_{:\tau}$ in the call-by-name evaluation strategy, then $[M]^{*n}_\tau \xrightarrow{*} [N]^{*n}_\tau$ in the call-by-name evaluation strategy.*

PROOF: Similar to Theorem 5. We give only the differences.

Since $\tau$ is a base type, $M_{:\tau}$ is either a variable, a basic constant or an application. As a result, after $*n$-normalization, the leftmost position of $[M]^{*n}_\tau$ corresponds to that of $M_{:\tau}$, so if we can reduce $M_{:\tau}$ in the call-by-name evaluation strategy, then we can also do $[M]^{*n}_\tau \to M'$ in it. By applying $*n$-normalization to the resulting $M'$, we obtain $[N]^{*n}_\tau$. □

**Theorem 6 (semantic correctness)** $[\cdot]^*_\tau$ *compilation is correct with respect to call-by-name operational semantics. That is, if $M_{:\tau}$ is a closed term of base type, then: (1) when call-by-name strategy leads to a normal form $N_{:\tau}$, then $[M]^*_\tau$ is reduced to the normal form $[N]^*_\tau$ by the same strategy; (2) when call-by-name strategy leads to an infinite reduction, its application to $[N]^*_{:\tau}$ also leads to an infinite reduction.*

PROOF: From Theorem 5 and confluence of both systems, we obtain (1): call-by-name leads to a normal form whenever it exists.

Since $[M]^{*n}_\tau \xrightarrow{*} [M]^*_\tau$, whenever call-by-name evaluation of $[M]^*_\tau$ leads to a normal form, evaluation of $[M]^{*n}_\tau$ leads to the same one. By Lemma 1, there exist $N_{:\tau}$, such that $[N]^*_\tau$ —in this case this is equal to $[N]^{*n}_\tau$— is this normal form. Once normalized by *Convert* and *Simplify*, $N_{:\tau}$ is also a normal form, and it is obtained by call-by-name normalization of $M_{:\tau}$. By negation of this, (2) stands. □