Research Report

ISG-RR-95-1

Compiling The Typed-Polymorphic Label-Selective λ-Calculus

Denys Duchier

May 1995

Abstract

In the label selective λ-calculus, arguments are passed by name rather than by position: abstractions $\lambda\{\ell \Rightarrow x\}\ M$ and applications $M\ \{\ell \Rightarrow N\}$ are parametrized by an explicit label ℓ and arbitrary commutations involving distinct labels are allowed. Is such an expressive language amenable to efficient execution? We show that, when $\Lambda^{\mathcal{S}}$ is equipped with an ML like type system the answer is *yes*: every well-typed $\Lambda^{\mathcal{S}}$ program can be transformed into an observationally equivalent ML program where all labels have been erased. Traditional compilation methods can then be applied.

Contents

1 Introduction 1

2 Embedding the λ-Calculus in $\Lambda^{\mathcal{S}}$ 1

3 A Record Calculus on $\mathbb{N}$ 2

4 $\Lambda^{\mathcal{S}}$ With Generalized Labels 2

5 The Compilation Challenge 3

6 The Typed Polymorphic $\Lambda^{\mathcal{S}}$ 3

7 Compiling $\Lambda^{\mathcal{S}}$ 3

8 Example 4

9 Interpretation Trick 5

10 Conclusion 5

1 Introduction

The prime innovation introduced by the label-selective λ-calculus $\Lambda^{\mathcal{S}}$ is to pass arguments by name rather than by position, as is normally the case is most programming languages. Common Lisp keywords afford an intermediate capability that permit arguments to be named and supplied in arbitrary order; however, this syntactic extension does not commute with currying. $\Lambda^{\mathcal{S}}$, on the other hand, is predicated on the commutation of named abstraction and application.

Positional Arguments: make_box(300,200)

Common Lisp Keywords: (make_box :width 300 :height 200)
(make_box :height 200 :width 300)

Label-Selective λ-Calculus: make_box $\{$width $\Rightarrow$ 300$\}$ $\{$height $\Rightarrow$ 200$\}$
make_box $\{$height $\Rightarrow$ 200$\}$ $\{$width $\Rightarrow$ 300$\}$

The syntax of $\Lambda^{\mathcal{S}}$ is that of the λ-calculus extended so that both abstraction and application are augmented with an explicit *label*:

$$M \quad ::= \quad x \mid \lambda\{\ell \Rightarrow x\}\ M \mid M\ \{\ell \Rightarrow M\}$$

Congruences. Named applications are allowed to commute, and so are named abstractions, provided the labels involved are distinct:

$$\begin{array}{rcl} f\ \{\ell_1 \Rightarrow e_1\}\ \{\ell_2 \Rightarrow e_2\} & \equiv & f\ \{\ell_2 \Rightarrow e_2\}\ \{\ell_1 \Rightarrow e_1\} \\ \lambda\{\ell_1 \Rightarrow x_1\}\lambda\{\ell_2 \Rightarrow x_2\}\ M & \equiv & \lambda\{\ell_2 \Rightarrow x_2\}\lambda\{\ell_1 \Rightarrow x_1\}\ M \end{array}$$

where $\ell_1 \neq \ell_2$ and $x_1 \neq x_2$.

Garrigue and Aït-Kaci [1, 2] introduce the additional congruence:

$$(\lambda\{\ell_1 \Rightarrow x_1\}\ M)\ \{\ell_2 \Rightarrow e_2\} \quad \equiv \quad \lambda\{\ell_1 \Rightarrow x_1\}\ (M\ \{\ell_2 \Rightarrow e_2\})$$

if $\ell_1 \neq \ell_2$ and x_1 is not free in e_2. They define a rewrite system based on these equations together with β-reduction and show its confluence.

Quotient Notation. In this paper, we will put forward a view based on congruence rather than one based on rewriting and normal forms. The syntax introduced earlier can be naturally extended to denote expressions in the quotient space:

$$\begin{array}{rcl} f\ \{\ell_1 \Rightarrow e_1, \ldots, \ell_n \Rightarrow e_n\} & \stackrel{\text{def}}{=} & f\ \{\ell_1 \Rightarrow e_1\} \ldots \{\ell_n \Rightarrow e_n\} \\ \lambda\{\ell_1 \Rightarrow x_1, \ldots, \ell_n \Rightarrow x_n\}\ M & \stackrel{\text{def}}{=} & \lambda\{\ell_1 \Rightarrow x_1\} \ldots \lambda\{\ell_n \Rightarrow x_n\}\ M \end{array}$$

where the ℓ_i are all distinct.

2 Embedding the λ-Calculus in $\Lambda^{\mathcal{S}}$

The most natural way to embed the λ-calculus in $\Lambda^{\mathcal{S}}$ is to use natural numbers as labels. Thus we define the syntactic equivalence:

$$f\ e_1\ e_2 \quad \stackrel{\text{def}}{=} \quad f\ \{1 \Rightarrow e_1, 2 \Rightarrow e_2\}$$

However, a look at currying shows that we must contend with an interesting new complication. Consider a function f of 2 curried arguments. A partial application of f to its first argument returns a new function that expects f's second argument as its first argument: in other words, it now expects that argument on label 1. Therefore, we must have the congruence:

$$\begin{array}{rl} & f\ \{1 \Rightarrow e_1, 2 \Rightarrow e_2\} \\ \equiv & f\ \{2 \Rightarrow e_2\}\ \{1 \Rightarrow e_1\} \\ \equiv & f\ \{1 \Rightarrow e_1\}\ \{1 \Rightarrow e_1\} \end{array}$$

In other words, it is necessary to renumber the labels.

3 A Record Calculus on $\mathbb{N}$

We now take a closer look at currying in $\Lambda^{\mathcal{S}}$ with integer labels. Since, in the quotient space, a function is applied to a record of arguments, the result of applying a function to several curried records must be the same as applying it to the combination of these records. Therefore, we shall now elucidate the rules governing record combination, also called *concatenation*.

Consider a function f of 4 arguments. The partial application of f to its 1st and 3rd arguments returns a function which expects f's 2nd and 4th arguments as its 1st and 2nd:

$$f\ \{1 \Rightarrow e_1, 2 \Rightarrow e_2, 3 \Rightarrow e_3, 4 \Rightarrow e_4\} \quad = \quad f\ \{1 \Rightarrow e_1, 3 \Rightarrow e_3\}\ \{1 \Rightarrow e_2, 2 \Rightarrow e_4\}$$

In the first partial application: the 2nd argument is the 1st missing and the 4th argument is the 2nd missing.

Record Concatenation. We shall write $r \star r'$ for the concatenation of records r and r', and it is defined as follows:

$$\begin{array}{rcl} r \star r' & = & \{i_1 \Rightarrow e_1, \ldots, i_n \Rightarrow e_n\} \star \{i'_1 \Rightarrow e'_1, \ldots, i'_p \Rightarrow e'_p\} \\ & = & \{i_1 \Rightarrow e_1, \ldots, i_n \Rightarrow e_n, \phi_r(i'_1) \Rightarrow e'_1, \ldots, \phi_r(i'_p) \Rightarrow e'_p\} \end{array}$$

where $\phi_r(i) = i$th position not used in r.

4 $\Lambda^{\mathcal{S}}$ With Generalized Labels

We may now combine the ideas of symbolic labels and integer labels: a generalized label is defined as a pair $\langle s, i\rangle$ of a symbol s and an integer i. A particularly interesting interpretation of such labels regards s as a channel name and i as a message number.

The natural embedding of the λ-calculus is now achieved by means of a distinguished channel name ι. Thus, integer label k is now represented by generalized label $\langle \iota, k\rangle$.

Generalized Congruence. The commutation congruences that take into account the necessary renumbering of labels can be expressed in the quotient space by the following equations:

$$\begin{array}{rcl} f\ r\ r' & \equiv & f\ r \star r' \\ \lambda r \lambda r'\ M & \equiv & \lambda r \star r'\ M \end{array}$$

where $\star$ is the natural extension to pairs $\langle s, i\rangle$ where renumbering is carried out separately for each channel name s as described earlier.

5 The Compilation Challenge

The question we shall consider in the remainder of the paper is at what cost does the expressiveness of $\Lambda^{\mathcal{S}}$ come? Consider these difficulties: arguments may be supplied out of order, as in $f\ \{\ell_2 \Rightarrow e_2\}\ \{\ell_1 \Rightarrow e_1\}$. More generally, a function can be partially applied to an argument out of order, e.g. $f\ \{\ell_2 \Rightarrow e_2\}$. It can even be applied to an argument that is not meant for itself, but for an eventual result: $f\ \{\ell_3 \Rightarrow e_3\}$. Finally, a function may be passed around as an argument and then invoked in unknown ways: $f\ \{\ell_1 \Rightarrow g\}$.

Can $\Lambda^{\mathcal{S}}$ be compiled and executed efficiently? In the following we show that, if we equip $\Lambda^{\mathcal{S}}$ with a type system, then the answer is a surprisingly simple yes.

6 The Typed Polymorphic $\Lambda^{\mathcal{S}}$

First, we extend $\Lambda^{\mathcal{S}}$ with a **let** construct to obtain an ML-like language:

$$M \quad ::= \quad x \mid \lambda\{\ell \Rightarrow x\}\ M \mid M\ \{\ell \Rightarrow M\} \mid \textbf{let}\ x = M\ \textbf{in}\ M$$

We equip this language with an ML-like type system:

$$\begin{array}{rcl} \tau & ::= & c \mid v \mid \{\ell \Rightarrow \tau\} \rightarrow \tau \\ \sigma & ::= & \tau \mid \forall \bar{v}\ \tau \end{array}$$

and the usual inference rules:

$$\textbf{(1)} \qquad \Gamma, x : \forall \bar{v}\ \tau \vdash x : [\bar{\tau}'/\bar{v}]\tau \qquad\qquad \textbf{(2)} \quad \frac{\Gamma \vdash M : \{\ell \Rightarrow \tau\} \rightarrow \tau' \quad \Gamma \vdash N : \tau}{\Gamma \vdash M\ \{\ell \Rightarrow N\} : \tau'}$$

$$\textbf{(3)} \quad \frac{\Gamma, x : \tau \vdash M : \tau'}{\Gamma \vdash \lambda\{\ell \Rightarrow x\}\ M : \{\ell \Rightarrow \tau\} \rightarrow \tau'} \qquad \textbf{(4)} \quad \frac{\Gamma \vdash M : \tau \quad \Gamma, x : \forall \bar{v}\ \tau \vdash N : \tau'}{\Gamma \vdash \textbf{let}\ x = M\ \textbf{in}\ N : \tau'}$$

where $\bar{v}$ are the free vars of τ not free in Γ

Congruence on Types. The congruence on terms induces a congruence on types: a functional term must have all types corresponding to every permutation of its arguments.

$$\{\ell_1 \Rightarrow \tau_1\} \rightarrow \ldots \rightarrow \{\ell_n \Rightarrow \tau_n\} \rightarrow \tau \quad \equiv \quad \{\ell_1 \Rightarrow \tau_1\} \star \ldots \star \{\ell_n \Rightarrow \tau_n\} \rightarrow \tau$$

Rule **(2)** must be applied modulo this congruence so that τ in the first premise can always be identified with τ in the second premise.

Type Inference Algorithm. It is sufficient to modify the usual algorithm so that it maintains functional types in canonical form $\{\ell_1 \Rightarrow \tau_1, \ldots, \ell_n \Rightarrow \tau_n\} \rightarrow \tau$ where τ is not functional.

7 Compiling $\Lambda^{\mathcal{S}}$

We are going to show that every typable $\Lambda^{\mathcal{S}}$ program can be converted to the ordinary λ-calculus where labels have been erased. Traditional methods of compilation can then be applied.

The basis of this transformation is that for every $\Lambda^{\mathcal{S}}$ program typable modulo the congruence on types, there exists an equivalent program typable without the congruence. The constructive proof is by transformation of the type inference proof tree. The notion of equivalence between programs that we consider is *observational equivalence*.

The only problematic rule is **(2)** for typing applications:

Mismatch between actual and formal parameters. Notice that τ appears in both premises of (**2**). Congruence may be required to identify the occurrence of τ in the left premise (i.e. the type of the formal argument on label ℓ of abstraction M) with its occurrence in the right premise (i.e. the type of the actual argument N).

Argument out of order. Notice that left premise and conclusion share the same label ℓ. It may well be that the type inferred for M does not naturally take its first argument on label ℓ but requires congruence to bring the argument on ℓ to the front.

Type driven syntactic transform. In either of the cases we just described, we have an expression whose inferred type is congruent but not identical to its desired type. We now define a type driven syntactic transform that maps an expression N of type τ to an equivalent expression $N|_{\tau}^{\tau'}$ of type τ'. The transformation uses η conversion to take arguments in the order in which they are given and supply them in the order in which they are expected.

$$\begin{aligned} N|_{\tau}^{\tau} &= N \\ N|_{\{\ell_1\Rightarrow\tau_1\}\to\cdots\to\{\ell_n\Rightarrow\tau_n\}\to\tau}^{\{\ell'_1\Rightarrow\tau'_1\}\to\cdots\to\{\ell'_n\Rightarrow\tau'_n\}\to\tau} &= \lambda\{\ell'_1 \Rightarrow x_1\}\ldots\lambda\{\ell'_n \Rightarrow x_n\}\; N\{\ell_1 \Rightarrow y_1|_{\tau''_1}^{\tau_1}\}\ldots\{\ell_n \Rightarrow y_n|_{\tau''_n}^{\tau_n}\} \end{aligned}$$

where y_i and τ''_i are defined by the congruences:

$$\begin{aligned} \{\ell'_1 \Rightarrow \tau'_1\} \star \cdots \star \{\ell'_n \Rightarrow \tau'_n\} &= \{\ell_1 \Rightarrow \tau''_1\} \star \cdots \star \{\ell_n \Rightarrow \tau''_n\} \\ \{\ell'_1 \Rightarrow x_1\} \star \cdots \star \{\ell'_n \Rightarrow x_n\} &= \{\ell_1 \Rightarrow y_1\} \star \cdots \star \{\ell_n \Rightarrow y_n\} \end{aligned}$$

8 Example

Consider the following program where $h\ r = M$ means $h = \lambda r\ M$:

$\lambda\ \{i \Rightarrow v\}$
  let g$\{a \Rightarrow x\}\{b \Rightarrow y\}$ = y **in**
  let f$\{c \Rightarrow x\}\{d \Rightarrow y\}$ = x$\{b \Rightarrow y\}$ **in**
    f$\{a \Rightarrow v\}\{d \Rightarrow v\}\{c \Rightarrow g\}$

We will not explicitate the whole typing tree; instead we will just look at the program's last line. It is clear that f and g have the following type schemes:

$$\begin{aligned} g &: \forall\alpha\beta.\{a \Rightarrow \alpha\} \to \{b \Rightarrow \beta\} \to \beta \\ f &: \forall\alpha\beta.\{c \Rightarrow \{b \Rightarrow \alpha\} \to \beta\} \to \{d \Rightarrow \alpha\} \to \beta \end{aligned}$$

Furthermore the inferred type for the occurrence of f on the last line is:

$$\{c \Rightarrow \{b \Rightarrow \alpha\} \to \{a \Rightarrow \alpha\} \to \alpha\} \to \{d \Rightarrow \alpha\} \to \{a \Rightarrow \alpha\} \to \alpha$$

whereas its desired type is:

$$\{a \Rightarrow \alpha\} \to \{d \Rightarrow \alpha\} \to \{c \Rightarrow \{a \Rightarrow \alpha\} \to \{b \Rightarrow \alpha\} \to \alpha\} \to \alpha$$

Therefore, we must replace the last line with its transform:

$$\begin{aligned} & f|_{\{c\Rightarrow\{b\Rightarrow\alpha\}\to\{a\Rightarrow\alpha\}\to\alpha\}\to\{d\Rightarrow\alpha\}\to\{a\Rightarrow\alpha\}\to\alpha}^{\{a\Rightarrow\alpha\}\to\{d\Rightarrow\alpha\}\to\{c\Rightarrow\{a\Rightarrow\alpha\}\to\{b\Rightarrow\alpha\}\to\alpha\}\to\alpha} \\ &= \lambda\{a \Rightarrow x\}\lambda\{d \Rightarrow y\}\lambda\{c \Rightarrow z\}\ f\{c \Rightarrow z|_{\{a\Rightarrow\alpha\}\to\{b\Rightarrow\alpha\}\to\alpha}^{\{b\Rightarrow\alpha\}\to\{a\Rightarrow\alpha\}\to\alpha}\}\{d \Rightarrow y\}\{a \Rightarrow x\} \\ &= \lambda\{a \Rightarrow x\}\lambda\{d \Rightarrow y\}\lambda\{c \Rightarrow z\}\ f\{c \Rightarrow \lambda\{b \Rightarrow x'\}\lambda\{a \Rightarrow y'\}\ z\{a \Rightarrow y'\}\{b \Rightarrow x'\}\}\{d \Rightarrow y\}\{a \Rightarrow x\} \end{aligned}$$

Thus we obtain the observationally equivalent program:

$\lambda\{i \Rightarrow v\}$ =
  let f$\{a \Rightarrow x\}\{b \Rightarrow y\}$ = y **in**
  let g$\{c \Rightarrow x\}\{d \Rightarrow y\}$ = x$\{b \Rightarrow y\}$ **in**
    $(\lambda\{a \Rightarrow x\}\lambda\{d \Rightarrow y\}\lambda\{c \Rightarrow z\}$
      f$\{c \Rightarrow \lambda\{b \Rightarrow x'\}\lambda\{a \Rightarrow y'\}\ z\{a \Rightarrow y'\}\{b \Rightarrow x'\}\}\ \{d \Rightarrow y\}\{a \Rightarrow x\})$
    $\{a \Rightarrow v\}\{d \Rightarrow v\}\{c \Rightarrow g\}$

9 Interpretation Trick

We are now going to show that labels can be erased. First we are going to plunge $\Lambda^{\mathcal{S}}$ into Standard ML. The trick is to interpret a label ℓ both as a type and a data constructor as if defined by:

$$\textbf{datatype}\ 'a\ \textbf{of}\ \ell = l\ \textbf{of}\ 'a$$

Now $\{\ell \Rightarrow e\}$ in an application $M\ \{\ell \Rightarrow e\}$ is interpreted as an application of the data constructor ℓ to expression e, and $\{\ell \Rightarrow x\}$ in an abstraction $\lambda\{\ell \Rightarrow x\}\ M$ is interpreted as pattern matching.

This interpretation yields an ordinary ML program which is typable with the ordinary ML type system. This program is observationally equivalent to the original $\Lambda^{\mathcal{S}}$ program and can be compiled using traditional techniques.

Erasing labels. Sum types with only one alternative can be erased. Values are packed by an application of a constructor ℓ before being handed to the receiving function which then immediately unpacks them by pattern matching. Therefore we can simply erase all labels.

10 Conclusion

We have shown that when $\Lambda^{\mathcal{S}}$ is equipped with a ML-like type system every program typable using the congruence on types can be transformed into an equivalent program typable without the congruence. The transformation relies on η-conversion:

$$f \equiv \lambda\{\ell \Rightarrow x\}\ f\{\ell \Rightarrow x\}$$

which preserves observational equivalence: only values of base types can be compared; functions are observationally equivalent iff they produce the same values on all inputs.

Finally, we described a mapping of $\Lambda^{\mathcal{S}}$ into Standard ML that allowed us to erase all labels and permitted the application of standard method of compilation.

Our transformation method introduces many abstractions. It may be remarked, however, that many of them are *non-escaping* because they occur only in functional position and do not require closures to be created on the heap. In fact, since they often just rearrange the order of arguments, they may sometimes be completely eliminated by the compiler.

References

[1] Hassan Aït-Kaci, Jacques Garrigue. Label-selective λ-calculus. Digital PRL Research Report 31, Paris, May 1993.

[2] Hassan Aït-Kaci, Jacques Garrigue. Label-selective λ-calculus: Syntax and Confluence.

[3] Hassan Aït-Kaci, Jacques Garrigue. The Typed Polymorphic Label-selective λ-calculus. Digital PRL Research Report 35, Paris, Oct 1993.