

A Weak Calculus with Explicit Operators for Pattern Matching and Substitution

Julien Forest

Laboratoire de Recherche en Informatique (CNRS URM 8623),
Bât 490, Université Paris-Sud,
91405 Orsay CEDEX, France.
Email: forest@lri.fr.

Abstract. In this paper we propose a **Weak Lambda Calculus** called λP_w having explicit operators for **Pattern Matching** and **Substitution**. This formalism is able to specify functions defined by cases via pattern matching constructors as done by most modern functional programming languages such as OCAML. We show the main property enjoyed by λP_w , namely subject reduction, confluence and strong normalization.

1 Introduction

In this paper we propose a **Weak Lambda Calculus** with **Pattern Matching** and **Explicit Substitution** called λP_w . The calculus λP_w is inspired by calculi of explicit substitutions [ACCL90, BBLRD96, BR95, Kes96] and by calculi of patterns [KPT96, CK99, CK]. The weak nature of λP_w allows us to denote variables by names without requiring α -conversion to implement correctly the notion of reduction.

Theoretical study of functional programming has been enriched by the introduction of typed λ -calculi, explicit substitutions [ACCL90, BBLRD96, BR95, Kes96] and pattern matching [KPT96, CK99, CK]. These three notions are the main ingredients of the formalism we propose in this paper to model typed functional languages with function definition by cases.

In the early thirties, Church proposed the λ -*calculus* as a general theory of functions and logic. Typed versions of λ -calculus were then defined by Curry and Church, they became the standard theoretical tool for defining and implementing *typed functional programming languages*. There is however an important gap between λ -calculus and modern functional languages:

- On one hand, the operation of substitution is not incorporated in the language level but is left as a meta-operation.
- On the other hand, most popular functional languages (resp. proof assistants) allow the definition of functions (resp. proofs) by cases via pattern-matching mechanisms, while λ -calculus does not incorporate at all these constructs.

The first problem is solved by incorporating the so-called explicit substitutions into the language used to implement functional programming. To do this, one simply adds a new construction to denote substitution and new reduction rules to describe the interaction between substitution and other constructors of the language. Many calculi with explicit substitutions [ACCL90, BBLRD96, BR95, Kes96] have been proposed in the literature, operational and logic properties of these calculi were extensively studied [Les94, CHL96, Kes96].

The second problem is solved by allowing abstractions of function not only with respect to variables but also with respect to patterns. Thus, the form of the arguments of a given function can be specified in a very precise form; for instance, a term having the form $\lambda\langle x, y \rangle. M$ specifies that the expected argument is a pair.

In the early nineties, Kesner, Tannen and Puel [KPT96] proposed a **Calculus of Pattern Matching** as a tool of theoretical study of pattern matching à la ML. In 1999, Kesner and Cerrito [CK99, CK] refined the ideas in [KPT96] and defined the calculus TPC_{ES} as a formalism with **explicit pattern matching** and **explicit substitution**. Other languages with explicit pattern matching, such as for example the ρ -calculus [CK98, CKL01], were recently proposed in the literature to model other programming paradigms.

The calculus presented in this paper, called λP_w , is a calculus with explicit pattern matching and explicit substitutions. This calculus is not designed as a *user level language* but as the *output calculus* of a pattern matching compilation algorithm. Such an algorithm is supposed to take a pattern matching function definition and to return an equivalent one where all the ambiguities between overlapping patterns have disappeared and where incomplete patterns definitions have been detected and completed. Such an hypothesis is not really restrictive since all the functional languages with pattern matching features ([Obj]) apply such an algorithm before evaluating programs.

The calculus λP_w is based on [CK99, CK], but has the following new features:

- λP_w is a **weak calculus** of explicit substitutions, that is, functions are lazily evaluated. To implement this correctly, substitutions are not allowed to cross lambda constructors - so that α -conversion is no more needed to achieve correct reduction of terms - and composition of substitutions is incorporated into the substitution language in order to guarantee confluence. The syntax of λP_w is based on the weak σ -calculus with names [CHL96] in contrast to TPC_{ES} , which is a **strong** calculus based on the substitution formalism with names called x [BR95, Blo95].
- In contrast to TPC_{ES} which treats “ordinary” substitutions explicitly but the so-called “sum” substitutions implicitly¹, λP_w treats *all* the substitutions as explicit. This choice makes the formalism (typing rules and typing

¹ In fact, the first version presented in [CK99] treats sum substitutions explicitly, but the revised and corrected version in [CK] only keeps ordinary substitution as an explicit operation by moving sum substitution to the meta-level.

reduction rules) and the proofs more involved than those in [CK99], but results in a complete and self-contained formalism which is able to describe different implementations of functional languages with pattern matching.

The formalism that we present in this paper enjoys all the classical properties of typed λ -calculi.

- It is **confluent** on all terms.
- It has the **subject reduction** property.
- It is **strongly normalizing** on all well-typed terms.

The paper is organized as follows. We will first give in Section 2 a formal definition of λP_w and give some basic properties such as *preservation of free variables* by reduction and confluence on all ground terms. We then introduce in Section 3 a typing system for λP_w and show that λP_w enjoys the *subject reduction* property. We finally show *strong normalization* of well-typed λP_w -terms in Section 4 before the conclusion given in Section 5.

2 Definition of λP_w

We first define the *raw* expressions of the calculus λP_w by giving three different sets to denote respectively raw terms, raw substitutions and raw sum terms. The notion of raw expression is refined by first defining the set of free variables of any raw expression which allows us to define (well-formed) expressions such as terms, substitutions and sum terms. Reduction rules of λP_w are given in Figure 1. These rules are showed to preserve free variables of expressions.

We fix two distinct infinite sets of variables: the set of *usual variables*, noted x, y, z, \dots , which are used to denote ordinary terms, and the set of *sum variables*, noted ξ, ψ, \dots , which are used to denote disjunction. We also fix two constants L and R and we use the notation K to denote indistinctly one or the other one. We will also use the notation T to denote indistinctly L, R or a sum variable.

Types of λP_w are given by the following grammar:

$$\begin{array}{ll}
 \text{(Types) } A ::= \iota & \text{Base type} \\
 | A \times A & \text{Product Type} \\
 | A + A & \text{Sum Type} \\
 | A \rightarrow A & \text{Functional Type}
 \end{array}$$

Patterns of λP_w are given by the following grammar:

$$\begin{array}{ll}
 \text{(Pattern) } P ::= _ & \text{Wildcard} \\
 | x & \text{Variable} \\
 | \langle P, P \rangle & \text{Pair} \\
 | @\langle P, P \rangle & \text{Contraction} \\
 | (P \mid_{\xi} P) & \text{Sum}
 \end{array}$$

The notations $_$, x and $\langle P, Q \rangle$ are standard while the notation $@\langle P, Q \rangle$ is similar (indeed more general) to the **as** constructor of Ocaml [Obj]. The pattern

$(P \mid_{\xi} Q)$ is used to specify two different structures P and Q (of types A and B) corresponding to a pattern of sum type $A + B$. The sum variable ξ appearing in a pattern $(P \mid_{\xi} Q)$ is used to propagate the result of any matching w.r.t the pattern $(P \mid_{\xi} Q)$ all along the term where this variable occurs.

Raw Substitutions of λP_w are given by the following grammar:

(Raw Substitution) $s ::= id$	Identity
	$(x/M).s$ Cons_usual_var
	$(\xi^{PA}/K).s$ Cons_sum_var
	$s \circ s$ Concatenation

In order to mark which branch of a sum pattern has been chosen we use special syntax that we called *sum terms*. A sum term is either a constant (there is one for each possible choice), a sum variable (no choice has been made), or a substituted sum variable (the full evaluation has not been made).

(Raw Sum Terms) $\Xi ::= \xi$	Sum Variable
	L Left Constant
	R Right Constant
	$\xi[s]$ Sum Substitution

We are now able to introduce λP_w -terms. The main difference between λ -calculus and pattern calculi is that the notation $\lambda x.M$ is generalized to $\lambda P.M$ where P is a pattern as given by the previous grammar. Thus, λP_w -terms are given by the following grammar:

(Raw Terms) $M ::= x$	Usual Variable
	$(M N)$ Application
	$\langle M, M \rangle$ Pair
	$\text{inl}_B(M)$ Left injection
	$\text{inr}_A(M)$ Right injection
	$[M \mid_{\xi} M]$ Case
	$[M \mid_{\Xi}^s M]$ Frozen Case
	$\lambda P:A.M$ Abstraction
	$M[s]$ Closure

All along the paper we may sometimes omit types from expressions in order to simplify the notation, but expressions are supposed to be as defined by this grammar. A *Case* constructor of the form $[M \mid_{\xi} N]$ is used to specify two different terms M and N corresponding respectively to two different patterns P and Q of a sum pattern $(P \mid_{\xi} Q)$ appearing somewhere in the program. The communication between the case constructor $[M \mid_{\xi} N]$ and its corresponding sum pattern $(P \mid_{\xi} Q)$ is achieved via the sum variable ξ . The introduction of the Frozen Case constructor is purely technical, the idea is to prevent reduction of the sub-term M (resp. N) inside a case constructor of the form $[M \mid_{\xi} N]$ where a left (resp. right) choice has been already made.

Example 1. A simple λP_w -term is $\lambda(x \mid_{\xi} y) : A + B. [\lambda y' : B. \langle x, y' \rangle \mid_{\xi} \lambda x' : A. \langle x', y \rangle]$. For a more interesting example let us suppose that we have encoded

the recursive type nat as a sum type, and that $(0 \mid_{\xi} S m)$ is a pattern of type nat representing either 0 or a positive natural number of the form $S m$. We refer the reader to [CK99] for more examples and more details about encoding of recursive types in the formalism of pattern calculi. Indeed, the following Ocaml [Obj] term:

```
match n with
| 0      -> 0
| (S m) -> m
```

is given by the λP_w -term $\lambda(0 \mid_{\xi} (S m)) : nat.[0 \mid_{\xi} m]$.

Definition 21 (Raw expression) A *raw expression* is either a raw term, a raw substitution or a raw sum term.

As usually done in λ -calculus we will work modulo α -conversion. This notion must be defined with care since bound variables are not only *all* the variables appearing in *complex* patterns, but also, *all* the variables bound by substitutions.

Definition 22 (Binding Variables) The set of *Binding Variables* of a pattern (resp. a substitution) is defined as:

$$\begin{array}{ll}
BVar(-) & = \emptyset \\
BVar(x) & = \{x\} \\
BVar(\langle P, Q \rangle) & = BVar(P) \cup BVar(Q) \\
BVar(\langle P \mid_{\xi} Q \rangle) & = BVar(P) \cup BVar(Q) \cup \{\xi\} \\
BVar(@\langle P, Q \rangle) & = BVar(P) \cup BVar(Q) \\
BVar(id) & = \emptyset \\
BVar((x/M).s) & = BVar(s) \cup \{x\} \\
BVar((\xi^P/K).s) & = BVar(s) \cup BVar(P) \cup \{\xi\} \\
BVar(s \circ t) & = BVar(s) \cup BVar(t)
\end{array}$$

We have for example, $BVar((x \mid_{\xi} y)) = \{\xi, x, y\}$ and $BVar((x/M).(\xi^y/L).id) = \{x, y, \xi\}$.

Definition 23 (Free Variables) The set of *Free Variables* of an expression e is given by:

$$\begin{array}{ll}
FV(x) & = \{x\} \\
FV(\text{inl}(M)) = FV(\text{inr}(M)) & = FV(M) \\
FV(M N) = FV(\langle M, N \rangle) & = FV(M) \cup FV(N) \\
FV(\lambda P.M) & = FV(M) \setminus BVar(P) \\
FV(M[s]) & = (FV(M) \setminus BVar(s)) \cup FV(s) \\
FV([M \mid_{\xi} N]) & = FV(M) \cup FV(N) \cup \{\xi\} \\
FV([M \mid_{\xi}^s N]) & = ((FV(M) \cup FV(N)) \setminus BVar(s)) \cup FV(s) \cup FV(\Xi) \\
FV(id) = FV(K) & = \emptyset \\
FV((x/M).s) & = FV(M) \cup FV(s) \\
FV((\xi^P/K).s) & = FV(s) \\
FV(s \circ t) & = (FV(s) \setminus BVar(t)) \cup FV(t) \\
FV(\xi) & = \{\xi\} \\
FV(\xi[s]) & = (\{\xi\} \setminus BVar(s)) \cup FV(s)
\end{array}$$

Thus for example, $FV(\lambda(x \mid_{\xi} y).[x \mid_{\psi} t]) = \{\psi, t\}$ and $FV([x \mid_{\xi}^{(\xi^y/L).id} y]) = \{x, t, \xi\}$.

We define the set of *free sum variables (FSV)* of a raw expression e as the set of *sum variables* of e which are in $FV(e)$.

Definition 24 (Bound variables) The *Bound Variables* of a raw expression e are those variables appearing in e but not free in e .

We are now ready to define α -conversion on λP_w -expressions as simply renaming of bound variables. Thus, for example $\lambda(x \mid_{\xi} x).x$ and $\lambda(y \mid_{\psi} y).y$ are α -equivalent, but neither $\lambda(x \mid_{\xi} x).y$ and $\lambda(y \mid_{\xi} y).y$ nor $\lambda(x \mid_{\xi} x).x$ and $\lambda(x \mid_{\xi} y).x$ are α -equivalent.

We are now ready to introduce the reduction rules which are given in Figure 1. The **Pattern matching** rules are the rules implementing the pattern matching.

The **Propagation of Substitutions, Substitutions and Variables and Constants, Substitutions and Composition** rules are a natural extension of those of the σ -calculus.

The **Case** rules explain the mechanism to distribute a substitution s with respect to a case term $[M \mid_{\xi} N]$ which consists in:

- We first transform the case term into a frozen case term using the rule (*Freeze*).
- We then treat the part $\xi[s]$ of the obtained frozen case term until a result (*i.e.* the variable ξ , the constants **L** or **R**) is obtained.
- We can then distribute the substitution in the frozen case term using one of the rules (*Left*), (*Right*) or (*Xi*).

The reduction system generated by the rules *Abs_id*, *Abs_pair*, *Abs_contr*, *Abs_left*, *Abs_right*, *Abs_var* and *Abs_wild* is used to implement the pattern matching operation and is noted by \longrightarrow_P . All the other rules generate the reduction system used to implement the behavior of substitution and is noted by \longrightarrow_{es} . The reduction relation $\longrightarrow_{\lambda P_w}$ is generated by $\longrightarrow_{es} \cup \longrightarrow_P$. To simplify the notation we may simply note \longrightarrow for $\longrightarrow_{\lambda P_w}$ in the rest of the paper.

Example 2. We show one way to propagate the substitution $s = (x/M_3).(\xi^{PA}/L).id$ inside the term $M = [M_1 \mid_{\xi} M_2]$.

- First of all we reduce the case term into a frozen case term by $M[s] \longrightarrow_{Freeze} [M_1 \mid_{\xi[s]}^s M_2]$
- We then "evaluate" the part $\xi[s]$: $\xi[s] \longrightarrow_{Sub_sum_var_4} \xi[(\xi^{PA}/L).id] \longrightarrow_{Sub_sum_var_2} \mathbf{L}$
- Thus we have $[M_1 \mid_{\xi[s]}^s M_2] \longrightarrow^+ [M_1 \mid_{\mathbf{L}}^s M_2]$, and thus applying the rule (*Left*), we obtain $M[s] \longrightarrow^+ M_1[s]$.

Start Rule		
$(\lambda P.M) N$	\longrightarrow	$(\lambda P.M)[id] N \quad (Abs_id)$
Pattern Matching		
$(\lambda \langle P_1, P_2 \rangle . M)[s] \langle N_1, N_2 \rangle$	\longrightarrow	$((\lambda P_1 . \lambda P_2 . M)[s] N_1) N_2 \quad (Abs_pair)$
$(\lambda @ (P_1, P_2) . M)[s] N$	\longrightarrow	$((\lambda P_1 . \lambda P_2 . M)[s] N) N \quad (Abs_contr)$
$(\lambda (P_1 \mid_{\xi} P_2) . M)[s] \mathbf{inl}(N)$	\longrightarrow	$(\lambda P_1 . M)[(\xi^{P_2} / \mathbf{L}).s] N \quad (Abs_left)$
$(\lambda (P_1 \mid_{\xi} P_2) . M)[s] \mathbf{inr}(N)$	\longrightarrow	$(\lambda P_2 . M)[(\xi^{P_1} / \mathbf{R}).s] N \quad (Abs_right)$
$(\lambda x . M)[s] N$	\longrightarrow	$M[(x/N).s] \quad (Abs_var)$
$(\lambda _ . M)[s] N$	\longrightarrow	$M[s] \quad (Abs_wild)$
Case		
$[M \mid_{\xi} N][s]$	\longrightarrow	$[M \mid_{\xi[s]}^s N] \quad (Freeze)$
$[M \mid_{\mathbf{L}} N]$	\longrightarrow	$M[s] \quad (Left)$
$[M \mid_{\mathbf{R}} N]$	\longrightarrow	$N[s] \quad (Right)$
$[M \mid_{\xi}^s N]$	\longrightarrow	$[M[s] \mid_{\xi} N[s]] \quad (Xi)$
Propagation of Substitutions		
$(MN)[s]$	\longrightarrow	$M[s]N[s] \quad (Sub_app)$
$\mathbf{inl}(M)[s]$	\longrightarrow	$\mathbf{inl}(M[s]) \quad (Sub_left)$
$\mathbf{inr}(M)[s]$	\longrightarrow	$\mathbf{inr}(M[s]) \quad (Sub_right)$
$\langle M_1, M_2 \rangle [s]$	\longrightarrow	$\langle M_1[s], M_2[s] \rangle \quad (Sub_pair)$
Substitutions and Variables and Constants		
$x[id]$	\longrightarrow	$x \quad (Sub_var_1)$
$x[(x/N).s]$	\longrightarrow	$N \quad (Sub_var_2)$
$y[(x/N).s]$	\longrightarrow	$y[s] \text{ if } y \neq x \quad (Sub_var_3)$
$x[(\xi^P/K).s]$	\longrightarrow	$x[s] \quad (Sub_var_4)$
$\xi[id]$	\longrightarrow	$\xi \quad (Sub_sum_var_1)$
$\xi[(\xi^P/K).s]$	\longrightarrow	$\mathbf{K} \quad (Sub_sum_var_2)$
$\xi[(\psi^P/K).s]$	\longrightarrow	$\xi[s] \text{ if } \xi \neq \psi \quad (Sub_sum_var_3)$
$\xi[(x/M).s]$	\longrightarrow	$\xi[s] \quad (Sub_sum_var_4)$
Substitutions and Composition		
$M[s][t]$	\longrightarrow	$M[s \circ t] \quad (Sub_clos)$
$(s \circ t) \circ u$	\longrightarrow	$s \circ (t \circ u) \quad (Sub_ass_env)$
$((x/M).s) \circ t$	\longrightarrow	$(x/M[t]).(s \circ t) \quad (Sub_concat_1)$
$((\xi^P/K).s) \circ t$	\longrightarrow	$(\xi^P/K).(s \circ t) \quad (Sub_concat_2)$
$id \circ s$	\longrightarrow	$s \quad (Sub_id)$

Fig. 1. Reduction Rules for λP_w

Remark 1. Let s and s' be raw substitutions such that $s \longrightarrow s'$, then $BVar(s) = BVar(s')$.

However, the reduction system in Figure 1 is not really correct in the sense that $\longrightarrow_{\lambda P_w}$ does *not* preserve free variables. This is shown by the following example:

Example 3. Let M be a term such that $FV(M) = \emptyset$ and let $U = (\lambda(x \mid_{\xi} y).x \text{ inr}(M))$. Then $U \longrightarrow^*_{\lambda P_w} x[(\xi^x/R).id] \longrightarrow_{\lambda P_w} x$ and $FV(U) = \emptyset$ but $FV(x) = \{x\}$.

In order to avoid this problem we restrict the set of raw expressions in order to guarantee that no new free variable does appear along reduction sequences. The notion of *acceptable* expression, or simply *expression*, is obtained via the introduction of the following concepts:

Definition 25 (Localized Free Variables) Given a sum variable ξ , a sum constant K and a raw expression e , we define the set of *localized free variables* of e w.r.t. ξ and K , written as $FV_{\xi}^K(e)$, as the subset of $FV(e)$ define exactly as for $FV(e)$ except for the following cases:

$$\begin{aligned}
FV_{\xi}^L([M \mid_{\xi} N]) &= FV_{\xi}^L(M) \cup \{\xi\} \\
FV_{\xi}^R([M \mid_{\xi} N]) &= FV_{\xi}^R(N) \cup \{\xi\} \\
FV_{\xi}^L([M \mid_{\xi}^s N]) &= FV_{\xi}^L(M[s]) \cup \{\xi\} \\
FV_{\xi}^R([M \mid_{\xi}^s N]) &= FV_{\xi}^R(N[s]) \cup \{\xi\} \\
FV_{\xi}^L([M \mid_{\xi[t]}^s N]) &= FV_{\xi}^L(M[s]) \cup \{\xi\} && \text{if } \xi \notin BVar(t) \\
FV_{\xi}^R([M \mid_{\xi[t]}^s N]) &= FV_{\xi}^R(M[s]) \cup \{\xi\} && \text{if } \xi \notin BVar(t)
\end{aligned}$$

Intuitively, $FV_{\xi}^L(e)$ (resp. $FV_{\xi}^R(e)$) contains all the free variables of e except those that are on the right (resp. left) part of the sum terms rooted by the sum variable ξ . Thus, for example, $FV_{\xi}^L(\lambda(x \mid_{\xi} y).[x \mid_{\xi} t]) = \emptyset$, $FV_{\xi}^L(\lambda y.[x \mid_{\xi} t]) = \{x\}$, and $FV_{\xi}^K(x[(\xi^x/L).id]) = \{x\}$

We are finally ready to define the notion of *acceptable expression* which will avoid the example of creation of new free variables introduced before.

Definition 26 (Acceptable Expression) The raw expression e is said to be *acceptable* (or just called an *expression*) iff $Acc(e)$, where $Acc(\)$ is the least congruence on expressions such that every variable and every constant is acceptable

and also the following requirements hold:

$Acc(\psi[s])$	if $\psi \notin BVar(Q)$	$\forall(\xi^Q/K) \in s$
$Acc(M[s])$	if $FV_\xi^K(M) \cap BVar(Q) = \emptyset$	$\forall(\xi^Q/K) \in s$
$Acc(s \circ t)$	if $FV_\xi^K(s) \cap BVar(Q) = \emptyset$	$\forall(\xi^Q/K) \in t$
$Acc(\lambda P.M)$	if $(FV_\xi^R(M) \cap BVar(Q_1)) \cup (FV_\xi^L(M) \cap BVar(Q_2)) = \emptyset$	$\forall(Q_1 \mid_\xi Q_2) \in P$
$Acc([M \mid_\xi^s N])$	if $Acc(M[s])$ and $Acc(N[s])$	
$Acc([M \mid_{\xi[t]}^s N])$	if $Acc(M[s])$ and $Acc(N[s])$ and $Acc(\xi[t])$	if $(\xi^Q/K) \notin t$
$Acc([M \mid_{\xi[t]}^s N])$	if $Acc(M[s])$ and $Acc(\xi[t])$	if $(\xi^Q/L) \in t$
$Acc([M \mid_L^s N])$	if $Acc(M[s])$	
$Acc([M \mid_{\xi[t]}^s N])$	if $Acc(N[s])$ and $Acc(\xi[t])$	if $(\xi^Q/R) \in t$
$Acc([M \mid_R^s N])$	if $Acc(N[s])$	

Thus for example, the term $\lambda(x \mid_\xi y).[x \mid_\xi t]$ is acceptable while $\lambda(x \mid_\xi y).[x \mid_\psi t]$ is not acceptable. Indeed $FV_\xi^R([x \mid_\psi t]) = \{x, t, \xi\}$ and $BVar(x) = \{x\}$ and thus by definition of $Acc()$ for abstractions we have that $\lambda(x \mid_\xi y).[x \mid_\psi t]$ is not acceptable. The reader may also remark that the terms U and $x[(\xi^x/L).id]$ given in Example 3 are neither acceptable.

We have to show now that this new notion of acceptable expression recently introduced is correct to prevent creation of new free variables along reduction sequences, that is, reduction preserves acceptable expressions and free variables.

Lemma 21 If e is an expression and $e \longrightarrow_{\lambda P_w} e'$ then

- e' is an expression
- $FV(e') \subseteq FV(e)$

We are now ready to state *confluence* of λP_w which guarantees that normal forms are unique.

Theorem 22 (Confluence for λP_w) λP_w is confluent.

Proof. The proof technique used to show the confluence property is the same as in [CHL96]. By lack of space we can not give here all the details of this proof but we refer the interested reader to [For02] for full technical explanations. The idea/scheme of the proof is the following.

1. We first show *strong normalization* and *confluence* of the system \longrightarrow_{es} . This allows us to work with *es-normal* forms of λP_w -expressions.
2. We then define a reduction system \longrightarrow_{aux} on *es-normal* forms and we show that \longrightarrow_{aux} is confluent.
3. We finally conclude by Hardin's Interpretation Lemma [CHL96] which relates confluence of $\longrightarrow_{\lambda P_w}$ with confluence of \longrightarrow_{aux} .

3 A typing system for λP_w

As λ -calculus is strictly contained in λP_w , then λP_w is not strongly normalizing. In order to obtain strong normalization of λP_w we define a typing system which is capable of associating types to terms, sum terms and substitutions in a given environment. While typing systems have been already studied for calculi with explicit substitutions [DG01] and also for calculi with patterns [KPT96, CK], no formalism in the literature exists to correctly type explicit choice sum terms. The typing system that we present in this section is shown to have the *subject reduction* property, that is, typing is preserved under reduction. The notion of acceptable expression (Definition 26) is essential to guarantee such a property.

We restrict now our attention to a special kind of patterns called *acceptable*. For that, we say that a pattern P is *linear* if and only if every variable occurs at most once in P . We define a *type environment* to be a pair $\Phi; \Gamma$ such that Φ is a *sum environment* defined as a set of pairs of the form $\xi : K$ and Γ is a *pattern environment* defined as a set of *typed patterns*, which are pairs of the form $P : A$. We say that a type environment $\Phi; \Gamma$ is *linear* if every variable occurs at most once in $\Phi; \Gamma$.

Definition 31 (Acceptable Patterns and Environments) The set of *acceptable patterns of type A* , denoted by $\mathcal{AP}(A)$, is defined to be the smallest set of *linear* patterns verifying the following properties: $_ \in \mathcal{AP}(A)$; $x \in \mathcal{AP}(A)$ for any variable x ; $@(P, Q) \in \mathcal{AP}(A)$ if $P \in \mathcal{AP}(A)$ and $Q \in \mathcal{AP}(A)$; $\langle P, Q \rangle \in \mathcal{AP}(B \times C)$ and $(P \mid_\xi Q) \in \mathcal{AP}(B + C)$ if $P \in \mathcal{AP}(B)$ and $Q \in \mathcal{AP}(C)$. The role of the notion of “acceptable patterns” is to prevent the (*wildcard*) typing rule (corresponding to (*weakening*) in logic) to introduce meaningless pattern expressions. This notion extends naturally to environments by defining $\Phi; \Gamma$ to be *acceptable* if and only if each pattern appearing in $\Phi; \Gamma$ is acceptable.

Thus for example the pattern $\langle x, y \rangle$ is linear but *is not* in $\mathcal{AP}(A + B)$ since a pair pattern is not compatible with a sum type.

We now introduce the typing rules for terms and substitutions (resp. for sum terms) in Figure 2 (resp. Figure 3) assuming that all the patterns and type environments appearing in these rules are acceptable.

In the rules (*Case*₁) and (*Frozenscase*₁), ξ is a fresh sum variable and $(P \mid_\xi Q)$ is linear. In the rule (*Wildcard*), $\Phi; P : A, \Gamma$ has to be linear. In the rule (*Proj*₁), all the x_i ’s are distinct usual variables. In the rules (*Proj*₂) and (*Nproj*), all the ξ_i ’s are distinct sum variables. In the rule (*Nproj*) ξ does not appear in Γ . In the rule (*App*), we require that N does not contain free sum variables. In the rules (*Sub_term*), (*Frozenscase*₁) and (*Frozenscase*₂) we require that s does not contain free sum variables. In the rule (*Sub_cons*₁) we require that M does not contain free sum variables. In the rules (*Sub_cons*₁), (*Sub_cons*₂), all contexts have to be linear.

We say that the term M (resp. the substitution s and the sum term Ξ) *has type A* (resp. *co-environment $\Phi'; \Gamma'$* and *sum type T*) in a type environment $\Phi; \Gamma$ if and only if there is a type derivation ending with $\Phi; \Gamma \vdash M : A$ (resp.

$$\begin{array}{c}
\frac{}{\Phi; x_1 : A_1, \dots, x_n : A_n \vdash x_i : A_i} \text{ (Proj}_1\text{)} \\
\\
\frac{\Phi; \Gamma \vdash M : A}{\Phi; \Gamma \vdash \text{inl}_B(M) : A + B} \text{ (+Right}_1\text{)} \quad \frac{\Phi; \Gamma \vdash M : B}{\Phi; \Gamma \vdash \text{inr}_A(M) : A + B} \text{ (+Right}_2\text{)} \\
\\
\frac{\Phi; \Gamma \vdash \xi \rightsquigarrow \xi \quad \Phi; P : B, \Gamma \vdash M : A \quad \Phi; Q : C, \Gamma \vdash N : A}{\Phi; (P \mid_\xi Q) : B + C, \Gamma \vdash [M \mid_\xi N] : A} \text{ (Case}_1\text{)} \\
\\
\frac{\Phi; \Gamma \vdash \xi \rightsquigarrow \kappa \quad \Phi; \Gamma \vdash M : A \quad \Phi; \Gamma \vdash N : A}{\Phi; \Gamma \vdash [M \mid_\xi N] : A} \text{ (Case}_2\text{)} \\
\\
\frac{\Phi; \Gamma \vdash \Xi \rightsquigarrow \xi \quad \Phi; P : A; \Gamma \vdash M[s] : C \quad \Phi; Q : B; \Gamma \vdash N[s] : C}{\Phi; (P \mid_\xi Q) : A + B, \Gamma \vdash [M \mid_\Xi N] : C} \text{ (Frozenscase}_1\text{)} \\
\\
\frac{\Phi; \Gamma \vdash \Xi \rightsquigarrow \kappa \quad \Phi; \Gamma \vdash M[s] : A \quad \Phi; \Gamma \vdash N[s] : A}{\Phi; \Gamma \vdash [M \mid_\Xi N] : A} \text{ (Frozenscase}_2\text{)} \\
\\
\frac{\Phi; P : A, Q : B, \Gamma \vdash M : C}{\Phi; \langle P, Q \rangle : A \times B, \Gamma \vdash M : C} \text{ (\times Left)} \quad \frac{\Phi; \Gamma \vdash M : A \quad \Phi; \Gamma \vdash N : B}{\Phi; \Gamma \vdash \langle M, N \rangle : A \times B} \text{ (\times Right)} \\
\\
\frac{\Phi; P : A, \Gamma \vdash M : B}{\Phi; \Gamma \vdash \lambda P : A. M : A \rightarrow B} \text{ (\rightarrow Right)} \quad \frac{\Phi; \Gamma \vdash M : A \rightarrow B \quad \Phi; \Gamma \vdash N : A}{\Phi; \Gamma \vdash (MN) : B} \text{ (App)} \\
\\
\frac{\Phi; P : A, Q : A, \Gamma \vdash M : B}{\Phi; @\langle P, Q \rangle : A, \Gamma \vdash M : B} \text{ (Layered)} \quad \frac{\Phi; \Gamma \vdash M : B}{\Phi; P : A, \Gamma \vdash M : B} \text{ (Wildcard)} \\
\\
\frac{}{\Phi; \Gamma \vdash \text{id} \triangleright \Phi; \Gamma} \text{ (Sub_axiom)} \quad \frac{\Phi; \Gamma \vdash t \triangleright \Phi'; \Gamma' \quad \Phi'; \Gamma' \vdash s \triangleright \Phi''; \Gamma''}{\Phi; \Gamma \vdash s \circ t \triangleright \Phi''; \Gamma''} \text{ (Sub_concat)} \\
\\
\frac{\Phi; \Gamma \vdash M : A \quad \Phi; \Gamma \vdash s \triangleright \Phi'; \Gamma'}{\Phi; \Gamma \vdash (x/M).s \triangleright \Phi'; x : A, \Gamma'} \text{ (Sub_cons}_1\text{)} \quad \frac{\Phi; \Gamma \vdash s \triangleright \Phi'; \Gamma'}{\Phi; \Gamma \vdash (\xi^{PA}/K).s \triangleright \xi : K, \Phi'; P : A, \Gamma'} \text{ (Sub_cons}_2\text{)} \\
\\
\frac{\Phi; \Gamma \vdash s \triangleright \Phi'; \Gamma' \quad \Phi'; \Gamma' \vdash M : A}{\Phi; \Gamma \vdash M[s] : A} \text{ (Sub_term)}
\end{array}$$

Fig. 2. Typing Rules for Terms and Substitutions

$$\begin{array}{c}
\frac{}{\xi_1 : \mathbb{K}_1, \dots, \xi_m : \mathbb{K}_m; \Gamma \vdash \xi_j \rightsquigarrow \mathbb{K}_j} \text{ (Proj2)} \quad \frac{\text{if } \forall i, \xi \neq \xi_i}{\xi_1 : \mathbb{K}_1, \dots, \xi_m : \mathbb{K}_m; \Gamma \vdash \xi \rightsquigarrow \xi} \text{ (Nproj)} \\
\\
\frac{}{\Phi; \Gamma \vdash \mathbf{L} \rightsquigarrow \mathbf{L}} \text{ (L)} \quad \frac{}{\Phi; \Gamma \vdash \mathbf{R} \rightsquigarrow \mathbf{R}} \text{ (R)} \\
\\
\frac{\Phi; \Gamma \vdash s \triangleright \Phi'; \Gamma' \quad \Phi'; \Gamma' \vdash \xi \rightsquigarrow \mathbf{T}}{\Phi; \Gamma \vdash \xi[s] \rightsquigarrow \mathbf{T}} \text{ (Sub_sum)}
\end{array}$$

Fig. 3. Typing Rules for Sum Terms

$\Phi; \Gamma \vdash s \triangleright \Phi'; \Gamma'$ and $\Phi; \Gamma \vdash \Xi \rightsquigarrow \mathbf{T}$). We say that the term M (resp. the substitution s and the sum term Ξ) is *well-typed* in $\Phi; \Gamma$ if and only if there is a type A such that M has type A in $\Phi; \Gamma$ (resp there is an environment $\Phi'; \Gamma'$ such that s has is co-environment $\Phi'; \Gamma'$ in $\Phi; \Gamma$ and there is a sum type \mathbf{T} such that Ξ has sum type \mathbf{T} in $\Phi; \Gamma$). We will make an abuse of notation by writing $\Phi; \Gamma \vdash M : A$ (resp. $\Phi; \Gamma \vdash s \triangleright \Phi'; \Gamma'$ and $\Phi; \Gamma \vdash \Xi \rightsquigarrow \mathbf{T}$) to indicate that M (resp. s and Ξ) has type A in $\Phi; \Gamma$.

First of all, we remark that for any substitution s , the co-environment of s contains its typing environment. This observation can be formalized as follows:

Remark 2. If $\Psi; \Delta \vdash s \triangleright \Phi; \Gamma$ then there exists Ψ' and Δ' such that $\Phi = \Psi\Psi'$ and $\Gamma = \Delta\Delta'$.

Also note that for any well-typed expression e in $\Phi; \Gamma$ all its free variables appear in $\Phi; \Gamma$.

An important property used in the subject reduction proof (Theorem 34) states that if an expression e is well-typed in a typing environment $\Phi; \Gamma$, then it is also well-typed in any "reasonable" typing environment containing $\Phi; \Gamma$.

Lemma 31 (Weakening for Environments)

- If $\Phi; \Gamma \vdash M : A$ then for all acceptable $\Psi; \Delta$ such that $BVar(\Psi; \Delta) \cap (BVar(\Phi; \Gamma) \cup FSV(M)) = \emptyset$, then $\Psi\Phi; \Gamma\Delta \vdash M : A$.
- If $\Phi; \Gamma \vdash s \triangleright \Phi'; \Gamma'$, then for all acceptable $\Psi; \Delta$ such that $BVar(\Psi; \Delta) \cap (BVar(\Phi; \Gamma) \cup BVar(s) \cup FSV(s)) = \emptyset$, then $\Psi\Phi; \Delta\Gamma \vdash s \triangleright \Psi\Phi'; \Delta\Gamma'$.
- If $\Phi; \Gamma \vdash \Xi \rightsquigarrow \mathbb{K}$, then for all acceptable $\Psi; \Delta$ such that $BVar(\Psi; \Delta) \cap (BVar(\Phi; \Gamma) \cap FSV(\Xi)) = \emptyset$, then $\Psi\Phi; \Delta\Gamma \vdash \Xi \rightsquigarrow \mathbb{K}$.

Proof. We prove these three statements by induction on $(|\Psi| + |\Delta|, h)$ where $|\Delta|$ is the number of patterns appearing in Δ , $|\Psi|$ is the number of sum variables appearing in Ψ and h is the height of the considered proof.

The proof of subject reduction is strongly based on the possibility of deconstructing patterns into more simpler ones via the $Dec()$ operation which is defined as follows:

Definition 32 Given a typed pattern $P : A$, we define its *deconstruction* as follows:

$$\begin{aligned}
Dec(_ : A) &= _ : A & Dec(x : A) &= x : A \\
Dec((P_1 \mid_{\xi} P_2) : A_1 + A_2) &= (P_1 \mid_{\xi} P_2) : A_1 + A_2 \\
Dec(\langle P_1, P_2 \rangle : A_1 \times A_2) &= Dec(P_1 : A_1), Dec(P_2 : A_2) \\
Dec(@ (P_1, P_2) : A) &= Dec(P_1 : A), Dec(P_2 : A)
\end{aligned}$$

This notion extends naturally to a pattern environment $\Gamma = P_1 : A_1, \dots, P_n : A_n$ by defining $Dec(\Gamma)$ as $Dec(P_1 : A_1), \dots, Dec(P_n : A_n)$.

The typing system enjoys the property that any well-typed expression in a typing environment is also a well-typed expression in the deconstructed environment.

Lemma 32 If e is well typed in an environment $\Phi; \Gamma$ then e is also well typed in $\Phi; Dec(\Gamma)$.

Another property of typing derivations which is needed in the proof of subject reduction states that for every well typed expression e we can choose a “canonical” typing derivation ending with a particular typing rule associated to e . Thus for example, if $\Phi; \Gamma \vdash \langle M_1, M_2 \rangle : A \times B$ then there is a proof of this sequent ending with the rule ($\times Right$). We refer the interested reader to [For02] for further details.

The deconstruction operation given in Definition 32 can be used to simplify pair and contraction patterns, but there is no operation to simplify sum patterns. There is however a property of typing derivations which allows us to decompose sum patterns appearing on the left hand side of sequents as follows:

Lemma 33 Let K and K' be L or R, then

- If $\Phi; (P \mid_{\xi} Q) : A + B, \Gamma \vdash M : C$ then so is $\xi : K, \Phi; P : A, Q : B, \Gamma \vdash M : C$.
- If $\Phi; (P \mid_{\xi} Q) : A + B, \Gamma \vdash s \triangleright \Phi'; (P \mid_{\xi} Q) : A + B, \Gamma'$ then so is $\xi : K, \Phi; P : A, Q : B, \Gamma \vdash s \triangleright \xi : K, \Phi'; P : A, Q : B, \Gamma'$.
- If $\Phi; (P \mid_{\xi} Q) : A + B, \Gamma \vdash \Xi \rightsquigarrow K'$ then $\xi : K, \Phi; P : A, Q : B, \Gamma \vdash \Xi \rightsquigarrow K'$.
- If $\Phi; (P \mid_{\xi} Q) : A + B, \Gamma \vdash \Xi \rightsquigarrow \psi$ then $\xi : K, \Phi; P : A, Q : B, \Gamma \vdash \Xi \rightsquigarrow \psi$.

We can now state the following result:

Theorem 34 (Subject reduction)

- If $\Phi; \Gamma \vdash M : A$ and $M \longrightarrow M'$ then $\Phi; \Gamma \vdash M' : A$.
- If $\Phi; \Gamma \vdash s \triangleright \Phi'; \Gamma'$ and $s \longrightarrow s'$ then $\Phi; \Gamma \vdash s' \triangleright \Phi'; \Gamma'$.
- If $\Phi; \Gamma \vdash \Xi \rightsquigarrow T$ (with $T \in \{L, R, \xi\}$) and $\Xi \longrightarrow \Xi'$ then $\Phi; \Gamma \vdash \Xi' \rightsquigarrow T$.

Proof. By induction on expressions, using Lemmas 33, 32 and 31.

4 Strong normalization for λP_w

This section is devoted to the proof of strong normalization of well-typed λP_w -terms. This proof is an adaptation of the one proposed by Ritter [Rit94] for a restricted version of λ_σ with de Bruijn indices where substitutions can only cross the leftmost outermost lambda. The scheme of our proof can be summarized as follows:

- We first define a calculus *modulo* an equational theory, noted $\lambda P_{w/\equiv}$.
- We then define the notion of *reducible term* and *reducible substitution* for $\lambda P_{w/\equiv}$ -expressions. We show that any reducible term (resp. substitution) is strongly normalizing.
- We show that the term $M[s]$ and the substitution $t \circ s$ are reducible for any reducible substitution s , well-typed term M and well-typed substitution t .
- We use the previous point to show that any well-typed $\lambda P_{w/\equiv}$ -expression is reducible and thus strongly normalizing.
- Finally, we deduce strong normalization for λP_w -expressions from strong normalization for $\lambda P_{w/\equiv}$ -expressions.

We start the proof by introducing the notion of *void substitutions* which are special substitutions which do not *really* change the pattern environment part of the their typing environment (i.e. which do not bind new usual variable).

Definition 41 The set VS of void substitutions is defined to be the smallest set of substitutions stable by concatenation such that:

- $id \in VS$
- $(\xi^{PA}/K).s \in VS$ if and only if $s \in VS$ and $BVar(P) = \emptyset$

Void substitutions enjoy the following properties:

Remark 3.

- Let s be a void substitution such that $s \longrightarrow s'$, then s' is a void substitution.
- Any void substitution is strongly normalizing.

Lemma 41 Let s be a substitution such that $\Psi; \Delta \vdash s \triangleright \Phi; \emptyset$, then we have:

- $\Delta = \emptyset$
- s is a void substitution

Proof. The first point holds by Remark 2 and the second one by contradiction.

4.1 Definition of $\lambda P_{w/\equiv}$

We can now define the notion of $\lambda P_{w/\equiv}$ -reduction modulo an equational theory.

Definition 42 The congruence \equiv on λP_w -terms is defined by:

$$(M \ N)[s] =_{\text{Sub_app}} M[s] \ N[s] \quad (s \circ t) \circ u =_{\text{Sub_ass_env}} s \circ (t \circ u)$$

$$M[s][t] =_{\text{Sub_clos}} M[s \circ t]$$

We will consider the reduction system $\lambda P_{w/\equiv}$, where $a \longrightarrow_{\lambda P_{w/\equiv}} b$ if and only if there exist a', b' such that $a \equiv a' \longrightarrow_{\mathcal{R}} b' \equiv b$, where $\mathcal{R} = \lambda P_w \setminus \{\text{Sub_app}, \text{Sub_ass_env}, \text{Sub_clos}\}$. This definition can also be interpreted as a reduction on *equivalence classes*, i.e., $[a] \longrightarrow_{\lambda P_{w/\equiv}} [b]$ if and only if $a' \longrightarrow_{\mathcal{R}} b'$, for $a' \in [a]$ and $b' \in [b]$. We may use indistinctly both interpretations according to the context.

We remark that by subject reduction (Theorem 34), the notion of well-typed $\lambda P_{w/\equiv}$ -terms is well-defined.

4.2 Strong normalization for $\lambda P_{w/\equiv}$

We are now able to introduce the notion of reducible expressions, which makes use of the following concept.

Definition 43 (Neutral terms and substitutions)

- A term is neutral if and only if it is neither of the forms $(\lambda x.N)[s]$, $\text{inr}_A(M)$, $\text{inl}_B(M)$ nor $\langle M_1, M_2 \rangle$.
- A substitution s is *neutral* if and only if it is not of the form $(x/M).t$.

Notation 42 Let M be a term.

- $\mathcal{P}_{A \times B}^1(M)$ denotes the term $(\lambda \langle x, - \rangle : A \times B.x)[id] \ M$ where x is a fresh variable.
- $\mathcal{P}_{A \times B}^2(M)$ denotes the term $(\lambda \langle -, x \rangle : A \times B.x)[id] \ M$ where x is a fresh variable.
- $\mathcal{S}_{A+B}(M)$ denotes the term $(\lambda(x \mid_{\xi} y) : A + B.[\langle x, w_2 \rangle \mid_{\xi} \langle w_1, y \rangle])[id] \ M$ where x, y, w_1, w_2 and ξ are fresh variables.

Definition 44 (Reducible terms and substitutions) The set of *reducible terms* for a given type in an environment $\Phi; \Gamma$ is defined by induction on types as follows:

$$\llbracket \iota \rrbracket_{\Phi; \Gamma} =_{def} \{M \mid \Phi; \Gamma \vdash M : \iota \text{ and } M \text{ is strongly normalizing}\}.$$

$$\llbracket A \rightarrow B \rrbracket_{\Phi; \Gamma} =_{def} \{M \mid \Phi; \Gamma \vdash M : A \rightarrow B \text{ and } \forall N \in \llbracket A \rrbracket_{\Phi; \Gamma \Delta}, (M \ N) \in \llbracket B \rrbracket_{\Phi; \Gamma \Delta}\} \text{ where } \Delta \text{ satisfies the condition of the point 1 of Lemma 31.}$$

$$\llbracket A \times B \rrbracket_{\Phi; \Gamma} =_{def} \{M \mid \Phi; \Gamma \vdash M : A \times B, \mathcal{P}_{A \times B}^1(M) \in \llbracket A \rrbracket_{\Phi; \Gamma} \text{ and } \mathcal{P}_{A \times B}^2(M) \in \llbracket B \rrbracket_{\Phi; \Gamma}\}$$

$$\llbracket A+B \rrbracket_{\Phi; \Gamma} =_{def} \{M \mid \Phi; \Gamma \vdash M : A+B \text{ and } \forall \text{ fresh variables } w_1, w_2, \mathcal{S}_{A+B}(M) \in \llbracket A \times B \rrbracket_{\Phi; w_1:A, w_2:B, \Gamma}\}$$

The set of *reducible substitutions for an environment $\Phi; \Gamma$ in an environment $\Psi; \Delta$* is defined as follows:

$$\llbracket \Phi; \Gamma \rrbracket_{\Psi; \Delta} =_{def} \{s \mid \Psi; \Delta \vdash s \triangleright \Phi; \Gamma \text{ and } \forall (x:A) \in \Gamma, x[s] \in \llbracket A \rrbracket_{\Psi; \Delta}\}$$

Reducible terms enjoy the following expected properties:

Lemma 43 For every type C the following statements hold:

1. If $M \in \llbracket C \rrbracket_{\Phi; \Gamma}$, then M is strongly normalizing.
2. If $\Phi; \Gamma \vdash (xM_1 \dots M_n) : C$ and $M_1 \dots M_n$ are strongly normalizing, then $(xM_1 \dots M_n) \in \llbracket C \rrbracket_{\Phi; \Gamma}$.
3. If $M \in \llbracket C \rrbracket_{\Phi; \Gamma}$ and $M \longrightarrow M'$ then $M' \in \llbracket C \rrbracket_{\Phi; \Gamma}$.
4. If M is a neutral of type C and all its one-step reducts are reducible expressions, then M is reducible.

Proof. The proof is done by induction on the type C .

Lemma 44 Let M be a term in $\llbracket A \rrbracket_{\Phi; \Gamma}$. For all acceptable environment $\Psi; \Delta$ satisfying the conditions of Point 1 of Lemma 31, M is in $\llbracket A \rrbracket_{\Phi\Psi; \Gamma\Delta}$

Proof. By induction on the type A .

We can now deduce from Lemma 43 (Point 2) the following property:

Corollary 1. *All the variables are reducible.*

As for terms, reducible substitutions also enjoy the following expected properties:

Lemma 45

1. If $s \in \llbracket \Phi; \Gamma \rrbracket_{\Psi; \Delta}$, then s is strongly normalizing.
2. If $s \in \llbracket \Phi; \Gamma \rrbracket_{\Psi; \Delta}$ and $s \longrightarrow s'$ then $s' \in \llbracket \Phi; \Gamma \rrbracket_{\Psi; \Delta}$.
3. If s is a neutral substitution such that $\Psi; \Delta \vdash s \triangleright \Phi; \Gamma$ and all its one-step reducts are reducible, then $s \in \llbracket \Phi; \Gamma \rrbracket_{\Psi; \Delta}$.

Proof. We prove the properties by cases on Γ . In the case of $\Gamma = \emptyset$ the proof is done by remarking that s is void. In the case of $\Gamma \neq \emptyset$, the proof is done by using Lemma 43.

Lemma 46 Let s be a substitution in $\llbracket \Phi; \Gamma \rrbracket_{\Psi; \Delta}$. For any acceptable environment $\Phi'; \Gamma'$ satisfying the conditions of point 2 of Lemma 31, s is in $\llbracket \Phi'\Phi; \Gamma'\Gamma \rrbracket_{\Phi'\Psi; \Gamma'\Delta}$

Proof. By Lemmas 31 and 44.

Since id is neutral, well-typed and has no reducts, then we can conclude the following by Lemma 45.

Corollary 2. *Let $\Phi; \Gamma$ be a valid environment. Then $id \in \llbracket \Phi; \Gamma \rrbracket_{\Phi; \Gamma}$.*

We are now ready to prove the state statement of this section which allows us to prove that any well-typed expression is reducible.

Theorem 47 Let $\Psi; \Delta$ and $\Phi; \Gamma$ be valid environments and s be a substitution in $\llbracket \Phi; \Gamma \rrbracket_{\Psi; \Delta}$.

- For every *substitution* t such that $\Phi; \Gamma \vdash t \triangleright \Phi'; \Gamma'$, we have $t \circ s$ is in $\llbracket \Phi'; \Gamma' \rrbracket_{\Psi; \Delta}$.
- For every *term* M such that $\Phi; \Gamma \vdash M : A$, we have $M[s]$ is in $\llbracket A \rrbracket_{\Psi; \Delta}$.

Proof. This proof can be done by induction on the structure of the (substitution/term) e . It uses some technical lemmas which we cannot present here by lack of space but which are fully detailed in [For02]. Intuitively, these lemmas state how to deduce reducibility for a given expression from the reducibility of its sub expressions. The *equivalence relation* is used, for example, to deduce reducibility for $M[s][t]$ from reducibility of $M[s \circ t]$.

By Theorem 47 and Corollary 2 we have that the term $M[id]$ and the substitution $t \circ id$ are reducible and thus by Lemmas 43 and 45 $M[id]$ and $t \circ id$ turns out to be strongly normalizing so that the following result holds.

Theorem 48 ($\lambda P_{w/\equiv}$ **strong normalization**) Any well typed $\lambda P_{w/\equiv}$ -expression is $\lambda P_{w/\equiv}$ strongly normalizing.

4.3 Strong normalization for λP_w

In order to conclude with the main theorem of this section we use the following standard property [FKP99].

Lemma 49 Let $A = \langle \mathcal{O}, R_1 \cup R_2 \rangle$ be an abstract reduction system such that:

- R_2 is strongly normalizing;
- there exists a reduction system $S = \langle \mathcal{O}', R' \rangle$ and translation \mathcal{T} from \mathcal{O} to \mathcal{O}' such that:
 - $a \longrightarrow_{R_1} b$ implies $\mathcal{T}(a) \longrightarrow_{R'} \mathcal{T}(b)$,
 - $a \longrightarrow_{R_2} b$ implies $\mathcal{T}(a) = \mathcal{T}(b)$.

Then for all term $a \in \mathcal{O}$ such that $\mathcal{T}(a)$ is R' -strongly normalizing we have that a is $(R_1 \cup R_2)$ -strongly normalizing.

Theorem 410 (λP_w **strong normalization**) Any well typed λP_w -expression is λP_w strongly normalizing

Proof. By Lemma 48 and Lemma 49, where $R' = \longrightarrow_{\lambda P_{w/\equiv}}$, R_2 is the reduction system engendered by the three rule *Sub_app*, *Sub_ass_env* and *Sub_clos*, R_2 is the reduction system engendered by the remaining rules and \mathcal{T} is the canonical projection from λP_w -expression to $\lambda P_{w/\equiv}$ -expression.

5 Conclusion

In this paper we have presented a weak calculus λP_w with explicit operations for pattern matching and substitution. Our formalism is successively inspired by [KPT96] and [CK]. In contrast to [KPT96], which treats substitutions and pattern-matching as meta-level operations, we have incorporated them into the syntax of the language by introducing appropriate reduction and typing rules. In contrast to [CK], we have eliminated the use of α -conversion (as our calculus is weak), and we have considered here a more powerful system of substitutions having composition. However, in our opinion, the major progress w.r.t the calculus TPC_{ES} presented in [CK] is that λP_w incorporates *sum replacement* as an *explicit* operation, where the calculus TPC_{ES} uses *meta-level* substitutions for sum variables. This step was one of the main goals of the formalism we propose in this work.

We have shown that the calculus λP_w enjoys all the classical properties of typed λ -calculi, namely it is confluent on all terms, it has the subject reduction property and it is strongly normalizing on all well-typed terms.

In the future, we would like to extend λP_w with *algebraic data types*, in order to cover more realistic functional programming languages, and with more general syntax for binding structures, as for example Klop's CRS [Klo80], in order to cover not only functional programming, but also other programming paradigms.

We would also like to incorporate to our formalism some ideas of the ρ -calculus [CK98] which deals with explicit *pattern matching* in a *rewriting* formalism. In particular, the λP_w -calculus implements a fix pattern-matching algorithm in contrast to ρ -calculus which can be parametrized with different matching algorithms.

Last, but not least, we are studying different evaluation strategies for λP_w , namely *lazy* and *eager* evaluators, which represent concrete implementations of functional languages via the more theoretical notion of reduction system proposed in this paper. We expect that this future work will allow us to provide a better explanation of the interaction between the operations of pattern matching and substitution.

Acknowledgement The author would like to thank his director Delia Kesner for the time spent to help him and for all the important remarks done on this article.

References

- ACCL90. M. Abadi, L. Cardelli, P.-L. Curien, and Jean-Jacques Lévy. Explicit substitutions. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 31–46, San Francisco, 1990.
- BBLRD96. Zine-El-Abidine Benaïssa, Daniel Briaud, Pierre Lescanne, and Jocelyne Rouyer-Degli. λv , a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 6(5), 1996.
- Blo95. Roel Bloo. Preservation of strong normalization for explicit substitutions. Technical Report TR95-08, TUE Computing Science Reports, Eindhoven University of Technology, 1995.

- BR95. Roel Bloo and Kristoffer Rose. Preservation of strong normalization in named lambda calculi with explicit substitution and garbage collection. In *Computing Science in the Netherlands*, pages 62–72. Netherlands Computer Science Research Foundation, 1995.
- CHL96. Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, March 1996.
- CK. Serenella Cerrito and Delia Kesner. Pattern matching as cut elimination. <ftp://ftp.lri.fr/LRI/articles/kesner/pm-as-ce.ps.gz>.
- CK98. Horatiu Cirstea and Claude Kirchner. ρ -calculus, the rewriting calculus. In *5th International Workshop on Constraints in Computational Logics*, 1998.
- CK99. Serenella Cerrito and Delia Kesner. Pattern matching as cut elimination. In Giuseppe Longo, editor, *Fourteenth Annual IEEE Symposium on Logic in Computer Science*, Trento, Italy, July 1999. IEEE Comp. Soc. Press.
- CKL01. Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. Matching Power. In Aart Middeldorp, editor, *Proceedings of RTA'2001*, Lecture Notes in Computer Science, Utrecht (The Netherlands), May 2001. Springer-Verlag.
- DG01. R. David and B. Guillaume. A λ -calculus with explicit weakening and explicit substitution. *Mathematical Structures in Computer Science*, 11(1), 2001.
- FKP99. Maria C.F. Ferreira, Delia Kesner, and Laurence Puel. Lambda-calculi with Explicit Substitutions Preserving Strong Normalization. *Applicable Algebra in Engineering Communication and Computing*, 9(4):333–371, 1999.
- For02. Julien Forest. A calculus of pattern matching and explicit substitution draft. Technical Report LRI-1313, 2002. <http://www.lri.fr/~forest/lpw.ps.gz>.
- Kes96. Delia Kesner. Confluence properties of extensional and non-extensional λ -calculi with explicit substitutions. In Harald Ganzinger, editor, *7th International Conference on Rewriting Techniques and Applications*, volume 1103 of *Lecture Notes in Computer Science*, pages 184–199, New Brunswick, NJ, USA, July 1996. Springer-Verlag.
- Klo80. Jan Willem Klop. *Combinatory Reduction Systems*. Mathematical Centre Tracts 127, Mathematisch Centrum, Amsterdam, 1980.
- KPT96. Delia Kesner, Laurence Puel, and Val Tannen. A Typed Pattern Calculus. *Information and Computation*, 124(1):32–61, January 1996.
- Les94. Pierre Lescanne. From $\lambda\sigma$ to $\lambda\nu$ a journey through calculi of explicit substitutions. In *Conference Record of the 21st Symposium on Principles of Programming Languages*, Portland, Oregon, 1994. ACM Press.
- Obj. The Objective Caml language. <http://www.ocaml.org/>.
- Rit94. E. Ritter. Normalisation for typed lambda calculi with explicit substitution. *Lecture Notes in Computer Science*, 832:295–??, 1994.