Pattern Matching as Cut Elimination

Serenella Cerrito Delia Kesner
CNRS and Laboratoire de Recherche en Informatique
Bât 490, Université de Paris-Sud
91405 Orsay Cedex, France
{serena,kesner}@lri.fr

Abstract

We present a typed pattern calculus with explicit pattern matching and explicit substitutions, where both the typing rules and the reduction rules are modeled on the same logical proof system, namely Gentzen sequent calculus for minimal logic. Our calculus is inspired by the Curry-Howard Isomorphism, in the sense that types, both for patterns and terms, correspond to propositions, terms correspond to proofs, and term reduction corresponds to sequent proof normalization performed by cut elimination. The calculus enjoys subject reduction, confluence, preservation of strong normalization w.r.t a system with meta-level substitutions and strong normalization for well-typed terms. As a consequence, it can be seen as an implementation calculus for functional formalisms defined with meta-level operations for pattern matching and substitutions.

1. Introduction

In this paper we propose a **T**yped **P**attern **C**alculus with **E**xplicit **S**ubsitutions, called TPC_{ES}, where both the typing rules and the reduction rules are modeled on the same logical proof system, namely sequent calculus for minimal logic. The calculus TPC_{ES} is inspired by Curry-Howard Isomorphism, in the sense that types, both for patterns and terms, correspond to propositions, terms correspond to proofs and term reduction corresponds to proof normalization. Since the logical proof system underlying typing and reduction rules is Gentzen sequent calculus rather than natural deduction, proof normalization is actually cut elimination. Pattern matching and substitution computations are embedded in the calculus, via an *explicit* encoding, rather than being meta-level operations, and correspond to sequences of cut elimination steps.

In the modelisation of functional programming via classical simply typed λ-calculus, the evaluation process is modeled by β-reduction and the replacement of formal parameters by actual arguments by *substitution*. In such a calculus, the substitution is a meta-level operation, executed by operators that are external to the language. In λ-calculi with *explicit substitutions*, as, for instance, [1, 5, 8, 26], and the pattern calculus TPC_{ES} that we propose here, the substitution is internalized and handled by symbols and reduction rules belonging to the syntax of the calculus itself. Explicit substitutions, by decomposing the β rule into finer steps, closer to what happens in environment machines, allow a better understanding of the execution models.

Pattern matching is modeled in TPC_{ES} by reduction of terms having the form $\texttt{let}\ M\ \texttt{be}\ P : A\ \texttt{in}\ N$ where M and N are terms and P is a *complex* pattern, while explicit substitution computation is modeled by reduction of terms having the form $N[x/M]$, where x is a *variable* pattern. In the first case, a reduction step "decomposes" the pattern $P : A$ and the corresponding term M and corresponds to a cut elimination step replacing a cut over a non atomic formula by cuts on its subformulae; for instance, matching of $\langle M_1, M_2\rangle$ with $\langle P_1, P_2\rangle : A_1 \times A_2$ reduces to matching M_1 with $P_1 : A_1$ and matching M_2 with $P_2 : A_2$ and corresponds to the cut elimination step replacing a cut over the formula $A_1 \wedge A_2$ by cuts on A_1 and A_2. The reader accustomed with cut elimination procedures will recognize that this corresponds exactly to a "principal-principal" cut case, called also "key case". In the second case, the reduction step executes the substitution of the variable in the term N, either by distributing it over N or else by physically replacing the variable x by the term M when N is a variable. Also in this case the reduction corresponds to a step in the procedure of eliminating cuts in sequent proofs (a "non key-case"). In general, in calculi with explicit substitutions, cut elimination allows to model the elimination of explicit substitutions in a given term. For example, let us consider the following cut elimination step:

$$\dfrac{\dfrac{\mathcal{D}}{\Gamma \vdash A} \qquad \Gamma, A \vdash A\ (axiom)}{\Gamma \vdash A}\ (cut) \implies \dfrac{\mathcal{D}}{\Gamma \vdash A}$$

Now, let us interpret proofs by terms and propositions by types as suggested by the Curry-Howard correspondence. We then get the following transformation of typing derivations:

$$\frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash x : A \; (proj)}{\Gamma \vdash x[x/M] : A} \; (sub) \implies \Gamma \vdash M : A$$

which suggests that the process of cut elimination consists in reducing the term $x[x/M]$ to the term M, exactly as in the $Var1$ reduction rule in [1, 5, 8, 26] defined by: $x[x/M] \longrightarrow M$.

As far as we know, our proposal is the first rational reconstruction of pattern matching in terms of *cut elimination*. Pattern matching is a very natural way to define functions (resp. proofs) by cases in functional languages (resp. proof editors). Definitions by cases via pattern matching are much simpler to write and to understand and this is the reason why they became so popular in functional languages such as Hope [9], SML [33], Miranda [37], Caml-Light [31] and Haskell [23], and proof editors such as Coq [11], PVS [36], HOL [22], LEGO [30] and ALF [3].

To illustrate the difference between pattern matching function definitions and ordinary definitions we give here two different (but equivalent) programs written in [31] to implement the function merge, which takes two lists and returns the list which alternates the elements of the first and the second one:

```
#let rec merge = function
  ([]       , l) -> l
 |(a::l as z,[]) -> z
 |(a::l   , b::m) -> a::(b::merge(l,m));;

#let rec merge = function(l,m) ->
 if l=[] then m
   else if m=[] then l
        else h(l)::(h(m)::merge(t(l),t(m)));;
```

The first program makes use of pattern matching while the other one does not. Therefore, the second program has to decompose a list l via the destructors h(l), which accesses the head of l, and t(l), which accesses the tail of l. The intensive use of destructors clutters the programs and is typical in languages without pattern matching features such as LISP. On the other hand, calculi modeling pattern matching allow to make abstractions of functions not only with respect to variables, but also with respect to complex patterns. Thus, the form of the arguments of a given function can be specified; for instance, a term having the form $\lambda\langle x, y\rangle : A_1 \times A_2.M$ will specify that the expected argument of the function is a pair.

As we said our calculus is inspired by the Curry-Howard isomorphism. Traditionally, the Curry-Howard isomorphism is used to explain the simply typed lambda calculus as a computational interpretation of the natural deduction proof system for intuitionistic propositional logic (actually, minimal logic, since there is no negation). The constructor terms of the simply typed lambda calculus correspond to those natural deduction proofs built using only the introduction rules. Now, in all the functional languages mentioned above, making use of pattern matching, the patterns have the same syntax as constructor terms, but operationally they are *dual* to them. There is one logical proof system in which this kind of duality is made clear: Gentzen's sequent calculus. Thus, we use sequent right rules for connectives (which are the same as the introduction rules of natural deduction) to build and type terms, left rules for connectives to build and type nested patterns, the sequent cut rule to model a general *let* construct (which is the starting point of computations), as well as an explicit substitution constructor, together with the sequent structural rules left contraction and left weakening to model the layered and wildcard patterns present in languages as ML or Haskell. In contrast to [27], which presents a calculus inspired from Gentzen sequent calculus that models programs with meta-level pattern-matching and substitution, we keep both operations as internal (or explicit). Also, the notion of reduction in [27] does not correspond to normalization of sequent proofs, but to normalization of proofs in natural deduction. The novelty of our approach is the design of a typed pattern calculus as a computational interpretation of the Gentzen sequent proofs, thereby allowing to model pattern matching via cut elimination.

We prove that the TPC_{ES} calculus enjoys the properties of subject reduction, confluence, and strong normalization for *well-typed* terms. Moreover, the calculus enjoys preservation of strong normalization w.r.t. another system TPC, which is a property on *all pure* terms (i.e. terms not having explicit substitutions). The study of this property in the framework of calculi with explicit substitutions has received much attention since Mellies [32] has shown that λ_σ-calculus [1] does not preserve strong normalization of λ-calculus. Under a quite natural definition of *value*, we show that it is possible to evaluate any given (functional) *program* (modeled by a TPC_{ES} *closed* term), to a *result* (modeled by a value).

The rest of the paper is organized as follows. In Section we introduce the syntax, the typing rules and the reduction rules of TPC_{ES}. We also establish the subject reduction property for TPC_{ES}. Section shows how to code simply typed λ calculus into TPC_{ES} and how to extend TPC_{ES} to recursive types. In Section we define another pattern calculus, TPC; proving some properties of TPC helps us to prove certain properties of TPC_{ES}, namely confluence and strong normalization, which are established in Section . In this last section we also establish the reductibility of closed typed TPC_{ES} terms to values. Finally, in Section , we conclude and we compare our approach with other works. Due to space limitations we do not give here full proofs, but they are available in [10].

Throughout the paper we will use standard notations from rewriting that we borrow from [15].

2. The Calculus TPC_{ES}

2.1. Raw Syntax

The set of *types* is defined by the following grammar:

$$A ::= \iota \mid A \times A \mid A + A \mid A \to A$$

where ι ranges over base types, and the symbols $\times$, $+$ and $\to$ define, respectively, product, sum and functional types.

We distinguish two disjoint sets of variables: *usual variables* noted $x, y, v, z \ldots$ and *sum variables* noted $\xi, \rho, \psi, \ldots$, etc. Sum variables play an essential role in patterns of disjunctive type and make it possible to define functions by cases.

(Raw) Patterns (P) are defined by the following grammar:

$$P ::= \_ \mid x \mid \sharp x \mid \langle P, P \rangle \mid (P \mid_\xi P) \mid @(P, P)$$

The patterns $\_$ is called *wildcard*, x is a *variable*, $\sharp x$ is an *arrow pattern*, $\langle P, P \rangle$ a *product pattern*, $(P \mid_\xi P)$ a *sum pattern* and $@(P, P)$ a *contraction pattern*.

(Raw) Terms (M) are defined by the following grammar:

$$\begin{array}{lll} M & ::= & x \mid \langle M, M \rangle \mid \mathtt{inl}_A(M) \mid \mathtt{inr}_A(M) \\ & & \mid M[x/M] \mid M[\xi/\mathtt{K}] \mid \lambda P{:}A.M \mid [M \mid_T M] \\ & & \mid \mathtt{let}\ M\ \mathtt{be}\ P{:}A\ \mathtt{in}\ M \\ & & \mid z\ \mathtt{of}\ M\ \mathtt{is}\ P{:}A\ \mathtt{in}\ M \\ & & \mid \lambda P{:}A.M\ \mathtt{of}\ M\ \mathtt{is}\ P{:}A\ \mathtt{in}\ M \end{array}$$

where $T \in \{\xi, \mathtt{L}, \mathtt{R}\}$, $\mathtt{K} \in \{\mathtt{L}, \mathtt{R}\}$, ξ is a sum variable, x et z are usual variables, and L and R are special constants (which will be used to instantiate sum variables).

The term x is called a *variable*, $\langle M, M \rangle$ is a *pair*, $\mathtt{inl}_A(M)$ and $\mathtt{inr}_A(M)$ are, respectively, *left* and *right injections*, $M[x/M]$ and $M[\xi/\mathtt{K}]$, are, respectively, *usual* and *sum explicit substitutions*, $[M \mid_T M]$ is a *case*, $\lambda P : A.M$ is a *lambda abstraction*, $\mathtt{let}\ M\ \mathtt{be}\ P{:}A\ \mathtt{in}\ M$ is a *let*, and, finally, terms of the form $U\ \mathtt{of}\ M\ \mathtt{is}\ P{:}A\ \mathtt{in}\ M$ are *applications*. The (usual) variable z in an application term of the form $z\ \mathtt{of}\ M\ \mathtt{is}\ P : A\ \mathtt{in}\ M$ is said to be a *functional variable* and is introduced by the $(\to left)$ typing rule. Both sum variables and functional variables are also called *communication variables*; the intuitive reason of such a terminology will be explained in Section . We use the notation $\mathcal{T}_{TPC_{ES}}$ to denote the set of *raw* terms defined above.

Free and bound occurrences of variables of terms are defined as usual, with the understanding that the terms of the form $\lambda P : A.M$, $U\ \mathtt{of}\ N\ \mathtt{is}\ P : A\ \mathtt{in}\ M$, $\mathtt{let}\ N\ \mathtt{be}\ P : A\ \mathtt{in}\ M$ (resp. $M[x/N]$ and $M[\xi/\mathtt{K}]$) define bindings whose scope is M for all the variables occurring in P (resp. for x and ξ). The meta-expression $M\{x \leftarrow N\}$ (resp. $M\{\xi \leftarrow \mathtt{K}\}$) denotes the term M where all the free occurrences of x (resp. of ξ) have been replaced by the term N (resp. by the constant K). We denote by $Var(P)$ the set of variables occurring in the pattern P and by $FV(M)$ the set of free variables occurring in the term M.

We work modulo α-conversion so that renaming of bound variables is systematically used. We distinguish one particular kind of free variables of a term M, which is the set of *free communication variables*, written $FCV(M)$. Thus, for example if $N = \lambda(x_1 \mid_\xi x_2).[x_1 \mid_\xi x_2]$, then the set of free communication variables of the term $z\ \mathtt{of}\ N\ \mathtt{is}\ w\ \mathtt{in}\ (w\ \mathtt{of}\ [z_1 \mid_\zeta z_2]\ \mathtt{is}\ y\ \mathtt{in}\ y)$ is $\{z, \zeta\}$. A formal definition of this notion can be found in [10].

A pattern P is said to be *linear* if every variable occurs at most once in P. The set of *acceptable patterns* of type A, denoted by $AP(A)$, is defined as the smallest set of *linear* patterns verifying the following properties: $\_ \in AP(A)$; $x \in AP(A)$ for any variable x; $\sharp x \in AP(B \to C)$ for any variable x; $@(P, Q) \in AP(A)$ if $P \in AP(A)$ and $Q \in AP(A)$; $\langle P, Q \rangle \in AP(B \times C)$ and $(P \mid_\xi Q) \in AP(B + C)$ if $P \in AP(B)$ and $Q \in AP(C)$. 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.

Typing judgments have the form $\Gamma \ \triangleright\ M : A$ where Γ, called a *pattern type assignment*, is a multi-set of elements of the form $P : A$ (where P is a pattern and A is a type), so the order of the patterns in a pattern type assignment is not relevant. The order of patterns in $@(P_1, P_2)$ is also irrelevant (see the $layered$ typing rule below). As for patterns, a pattern type assignment Γ is said to be *linear* if every variable occurs at most once in Γ.

2.2. Typing Rules

The syntax for terms and patterns that we have introduced in Section is referred to as *raw*, to emphasize the fact that it may or may not type-check. For example, raw patterns are not necessarily linear but the well-typed ones are.

Typing rules are presented in Figure 1. In the (sub) typing rule, the notation $?v$ stands for a variable pattern v or an arrow pattern $\sharp v$. All the pattern type assignments that can be built using the typing rules are linear. In the $(proj)$ rule the x_j's are all distinct. In the $(+left)$ rule ξ is a fresh sum variable and $(P \mid_\xi Q)$ is a linear pattern. In the $(\to left)$ rule z is a fresh variable. In the $(\to left)$, (app), (let) and (sub) rules we require the condition $FCV(M) = \emptyset$. In

$(wildcard)$, we require $P \in AP(A)$ and $P : A, \Gamma$ to be linear. In the (*app*) rule we also ask $FCV(\lambda P.J) = \emptyset$.

Our typing rules are designed in the spirit of Curry-Howard isomorphism by assigning patterns (resp. terms) to the types occurring in left (resp. right) hand side of a sequent. Therefore, the *logical* rules corresponding to our calculus are those of minimal logic that can be obtained by erasing the patterns and terms in the judgments. In particular, the logical system behind our type system contains weakening, contraction, and cut rules corresponding, respectively, to the $(wildcard)$, $(layered)$, (let) and (sub) typing rules. Essentially, any right sequent rule of minimal logic for a connective # corresponds to a typing rule introducing a *term* of a specific type $\tau_{\#}$ in TPC_{ES} calculus, while any left sequent rule for a connective # corresponds to a typing rule introducing a *pattern* of type $\tau_{\#}$. Note, however, that $(+left)$ and $(\rightarrow left)$ do introduce both a pattern of type $\tau_{\#}$ and a term.

Note also that the (*app*) typing rule, allowing to introduce a term expressing the application of a function to an argument (see Section), corresponds to a *derived* rule in the logic, obtained by applying first a left implication rule then a cut rule. Such a rule is necessary in order to close the set of typable terms under reduction, however it is not derivable from the others in the typing system. In fact, combining a $(\rightarrow left)$ and a (let) rule the term $\mathtt{let}\ \lambda P.J\ \mathtt{be}\ \sharp z\ \mathtt{in}\ z\ \mathtt{of}\ M\ \mathtt{is}\ Q\ \mathtt{in}\ N$ is typable, but such a term reduces to the term $\lambda P.J\ \mathtt{of}\ M\ \mathtt{is}\ Q\ \mathtt{in}\ N$ (see Section) which is not typable in the absence of the (*app*) rule.

The cut rule itself corresponds to two distinct typing rules, introducing, respectively, let terms and (usual) explicit substitutions terms. Indeed, reduction rules for let terms will allow to express the pattern matching process *inside* the TPC_{ES} calculus, while reduction rules for explicit substitutions terms will allow to express the behavior of explicit substitutions as is classically done in the literature [1, 5, 8, 26]. Therefore, both pattern matching and explicit substitution computations can be modeled via the cut elimination process for sequent proofs. We will come back on this important point once the reduction rules will be presented, and we will further explain why the cut rule is interpreted via two different term constructors in the calculus.

The typing rules $(layered)$ and $(wildcard)$, corresponding, respectively, to the left structural rules contraction and weakening of minimal logic, allow to model the `as` and `wildcard` patterns largely used in programming languages such ML or Caml-Light. Therefore, we prefer to keep weakening in the logical calculus, even if it can be derived from the other logical rules we are considering. Note that the $(wildcard)$ typing rule allows to introduce not only the wildcard pattern $\_ : A$ (as it is done in [27]), but, more generally, *any* pattern $P : A$ generated by the pattern grammar, provided that $P \in AP(A)$ and that the pattern type assignment stays linear. The role of the condition $P \in AP(A)$ is to prevent the introduction of "garbage" pattern-like expressions as, for instance, $\langle x, y\rangle : B \rightarrow C$, which would be obviously meaningless. The fact that the $(wildcard)$ introduces syntax generated by rules that are external to the typing system corresponds exactly to the behavior of the weakening rule in logic, which introduces a formula generated by an external grammar for formulae.

The typing rules $(\mathtt{L})$, $(\mathtt{R})$, $(sub_{\mathtt{L}})$ and $(sub_{\mathtt{R}})$ correspond to a trivial derived sequent rule which derives a sequent $\Gamma \vdash A$ from the same sequent $\Gamma \vdash A$, and they have a purely technical role. Roughly, such rules allow to type terms obtained by matching a sum pattern with a term having a sum type (see the reduction rules $Case1$, $Case2$, $Dcase1$ and $Dcase2$ in Section). Thus, also these typing rules allow to close the set of typable terms w.r.t. substitutions generated by the reduction rules.

Note also that the formulation of the sequent rules that we have chosen is *additive*.

As we observed, the $(+left)$ rule and the $(\rightarrow left)$ rule introduce both a pattern on the left hand side of a typing judgement and a term on the right hand side. Also, they introduce, respectively, a fresh sum variable ξ and a fresh functional variable z, whose role is to establish a link between the two sides of the judgement, which explaines why both sum variables and functional variables are called "communication variables".

We point out that the condition on free communication variables for the typing rules $(\rightarrow left)$, (let), (sub) and (*app*), although it may seem restrictive, does not prevent the full encoding of the usual functions defined by pattern matching which are used in functional programming. Moreover, simply typed terms can be encoded in our calculus TPC_{ES}, as shown in Section . In the absence of such a condition, the set of typable terms would not be closed under reduction. Indeed, the following typing judgement $(y \mid_{\xi} y) : A + A \triangleright \mathtt{let}\ [y \mid_{\xi} y]\ \mathtt{be}\ x : A\ \mathtt{in}\ [x \mid_{\xi} x] : A$ would be provable, while the judgement $(y \mid_{\xi} y) : A + A \triangleright [[y \mid_{\xi} y] \mid_{\xi} [y \mid_{\xi} y] : A]$, where the term in the right hand side is obtained by reduction of the corresponding term in the first judgement (see Section), would not be so. The condition required in rules $(\rightarrow left)$, (let), (sub) and (*app*), even if omitted in [27], is also essential to recover the subject reduction property in a framework with meta-level substitutions.

Finally, we note that here, as opposed to the simply typed lambda calculus, a typing judgment may have several derivations.

All along this paper the type decorations may be omitted from the syntax to avoid cluttering the notation. Also, by an abuse of language we will write $\Gamma \triangleright M : A$ to say that

$$x_1 : A_1, \dots x_n : A_n \triangleright x_i : A_i \quad (proj)$$

$$\frac{P : A, \Gamma \triangleright M : C \qquad Q : B, \Gamma \triangleright N : C}{(P \mid_\xi Q) : A + B, \Gamma \triangleright [M \mid_\xi N] : C} \quad (+left)$$

$$\frac{\Gamma \triangleright \lambda P : A.J : A \to B \quad \Gamma \triangleright M : A \quad \Gamma, Q : B \triangleright N : C}{\Gamma \triangleright \lambda P : A.J \ \mathtt{of}\ M \ \mathtt{is}\ Q : B \ \mathtt{in}\ N : C} \quad (app)$$

$$\frac{P : A, Q : B, \Gamma \triangleright M : C}{\langle P, Q \rangle : A \times B, \Gamma \triangleright M : C} \quad (\times left) \qquad \frac{\Gamma \triangleright M : A \quad \Gamma \triangleright N : B}{\Gamma \triangleright \langle M, N \rangle : A \times B} \quad (\times right)$$

$$\frac{\Gamma \triangleright M : A}{\Gamma \triangleright \mathtt{inl}_B(M) : A + B} \quad (+right1) \qquad \frac{\Gamma \triangleright N : B}{\Gamma \triangleright \mathtt{inr}_A(N) : A + B} \quad (+right2)$$

$$\frac{P : A, \Gamma \triangleright M : B}{\Gamma \triangleright \lambda P : A.M : A \to B} \quad (\to right) \qquad \frac{\Gamma \triangleright M : A \quad Q : B, \Gamma \triangleright N : C}{\sharp z : A \to B, \Gamma \triangleright z \ \mathtt{of}\ M \ \mathtt{is}\ Q : B \ \mathtt{in}\ N : C} \quad (\to left)$$

$$\frac{\Gamma \triangleright M : A \quad P : A, \Gamma \triangleright N : B}{\Gamma \triangleright \mathtt{let}\ M \ \mathtt{be}\ P : A \ \mathtt{in}\ N : B} \quad (let) \qquad \frac{\Gamma \triangleright M : A \quad ?v : A, \Gamma \triangleright N : B}{\Gamma \triangleright N[v/M] : B} \quad (sub)$$

$$\frac{\Gamma \triangleright M : B}{P : A, \Gamma \triangleright M : B} \quad (wildcard) \qquad \frac{P_1 : A, P_2 : A, \Gamma \triangleright M : B}{@(P_1, P_2) : A, \Gamma \triangleright M : B} \quad (layered)$$

$$\frac{\Gamma \triangleright M : A}{\Gamma \triangleright [M \mid_{\mathtt{L}} N] : A} \quad (\mathtt{L}) \qquad \frac{\Gamma \triangleright M\{\xi \leftarrow \mathtt{L}\} : D}{\Gamma \triangleright M[\xi/\mathtt{L}] : D} \quad (sub_{\mathtt{L}})$$

$$\frac{\Gamma \triangleright N : A}{\Gamma \triangleright [M \mid_{\mathtt{R}} N] : A} \quad (\mathtt{R}) \qquad \frac{\Gamma \triangleright M\{\xi \leftarrow \mathtt{R}\} : D}{\Gamma \triangleright M[\xi/\mathtt{R}] : D} \quad (sub_{\mathtt{R}})$$

Figure 1. Typing Rules of TPC_{ES}

the judgment $\Gamma \triangleright M : A$ is provable.

2.3. Reduction Rules

In what follows we use $M[\theta]$ to denote either $M[x/N]$ or $M[\xi/\mathtt{K}]$. The *domain* of θ, written $Dom(\theta)$ is x (resp. ξ). The reduction rules of TPC_{ES} appears in Figure 2.

Remark 2.1 The reduction rule $Lambda$ is the only rule that generates a substitution of the form $[z/N]$ where z is a usual communication variable. Thus, such substitutions always verify that N is a λ-abstraction.

The subsystem of TPC_{ES} reduction rules containing only: $Var1$, $Var2$, $Dcut$, $Dpair$, $Dlambda$, $Dapp1$, $Dapp2$, $Dapp3$, $Dinl$, $Dinr$, $Dcase1$, $Dcase2$ is called ES.

The rules in $TPC_{ES} \setminus$ ES handle pattern matching as cut elimination, while the ES rules handle the explicit substitution of one variable, either by distributing it over a term N (this is the case of the DN rules), or else by actually performing it when N is a variable (this is the case of the $Var1$ and $Var2$ rules). It is apparent that pattern matching and explicit substitution are both forms of substitutions. However, while the first group of rules "decomposes" a *complex* pattern $P : A$ and the corresponding term M, the ES rules execute the substitution of a *variable* pattern (or a sum communication variable) in a term N, the Sub rule making the bridge between the two groups. This is the reason why the *cut* sequent rule is associated to two distinct term constructors.

Note that *pattern matching simulates cut elimination*.

$$
\begin{array}{lll}
(\lambda P.J)\ \mathtt{of}\ M\ \mathtt{is}\ Q\ \mathtt{in}\ N & \longrightarrow_{Of} & \mathtt{let}\ (\mathtt{let}\ M\ \mathtt{be}\ P\ \mathtt{in}\ J)\ \mathtt{be}\ Q\ \mathtt{in}\ N \\
\mathtt{let}\ \langle M_1, M_2\rangle\ \mathtt{be}\ \langle P_1, P_2\rangle\ \mathtt{in}\ N & \longrightarrow_{Pair} & \mathtt{let}\ M_1\ \mathtt{be}\ P_1\ \mathtt{in}\ (\mathtt{let}\ M_2\ \mathtt{be}\ P_2\ \mathtt{in}\ N) \\
\mathtt{let}\ M\ \mathtt{be}\ @(P_1, P_2)\ \mathtt{in}\ N & \longrightarrow_{Cont} & \mathtt{let}\ M\ \mathtt{be}\ P_1\ \mathtt{in}\ (\mathtt{let}\ M\ \mathtt{be}\ P_2\ \mathtt{in}\ N) \\
\mathtt{let}\ \mathtt{inl}(L)\ \mathtt{be}\ (P \mid_\xi Q)\ \mathtt{in}\ M & \longrightarrow_{Case1} & \mathtt{let}\ L\ \mathtt{be}\ P\ \mathtt{in}\ M[\xi/\mathtt{L}] \\
\mathtt{let}\ \mathtt{inr}(L)\ \mathtt{be}\ (P \mid_\xi Q)\ \mathtt{in}\ M & \longrightarrow_{Case2} & \mathtt{let}\ L\ \mathtt{be}\ Q\ \mathtt{in}\ M[\xi/\mathtt{R}] \\
\mathtt{let}\ (\lambda P.M)\ \mathtt{be}\ \sharp z\ \mathtt{in}\ N & \longrightarrow_{Lambda} & N[z/\lambda P.M] \\
\mathtt{let}\ M\ \mathtt{be}\ x\ \mathtt{in}\ N & \longrightarrow_{Sub} & N[x/M] \\
\mathtt{let}\ M\ \mathtt{be}\ \_\ \mathtt{in}\ N & \longrightarrow_{Weak} & N \\
[M \mid_{\mathtt{L}} N] & \longrightarrow_{\mathtt{L}} & M \\
[M \mid_{\mathtt{R}} N] & \longrightarrow_{\mathtt{R}} & N \\
x[x/M] & \longrightarrow_{Var1} & M \\
y[\theta] & \longrightarrow_{Var2} & y,\ \text{if } y \neq Dom(\theta) \\
(\mathtt{let}\ M\ \mathtt{be}\ P\ \mathtt{in}\ N)[\theta] & \longrightarrow_{Dcut} & \mathtt{let}\ M[\theta]\ \mathtt{be}\ P\ \mathtt{in}\ N[\theta] \\
\langle M, N\rangle[\theta] & \longrightarrow_{Dpair} & \langle M[\theta], N[\theta]\rangle \\
(\lambda P.M)[\theta] & \longrightarrow_{Dlambda} & \lambda P.M[\theta] \\
(z\ \mathtt{of}\ M\ \mathtt{is}\ P\ \mathtt{in}\ N)[z/L] & \longrightarrow_{Dapp1} & L\ \mathtt{of}\ M\ \mathtt{is}\ P\ \mathtt{in}\ N \\
(z\ \mathtt{of}\ M\ \mathtt{is}\ P\ \mathtt{in}\ N)[\theta] & \longrightarrow_{Dapp2} & z\ \mathtt{of}\ M[\theta]\ \mathtt{is}\ P\ \mathtt{in}\ N[\theta],\ \text{if } z \neq Dom(\theta) \\
(\lambda Q.J\ \mathtt{of}\ M\ \mathtt{is}\ P\ \mathtt{in}\ N)[\theta] & \longrightarrow_{Dapp3} & \lambda Q.J[\theta]\ \mathtt{of}\ M[\theta]\ \mathtt{is}\ P\ \mathtt{in}\ N[\theta] \\
\mathtt{inl}(L)[\theta] & \longrightarrow_{Dinl} & \mathtt{inl}(L[\theta]) \\
\mathtt{inr}(L)[\theta] & \longrightarrow_{Dinr} & \mathtt{inr}(L[\theta]) \\
[M \mid_T N][\theta] & \longrightarrow_{Dcase1} & [M[\theta] \mid_T N[\theta]],\ \text{if } T \neq Dom(\theta) \\
[M \mid_\xi N][\xi/\mathtt{K}] & \longrightarrow_{Dcase2} & [M[\xi/\mathtt{K}] \mid_{\mathtt{K}} N[\xi/\mathtt{K}]]
\end{array}
$$

Figure 2. Reduction Rules of TPC_{ES}

Consider the $\longrightarrow_{Pair}$ reduction rule, for instance. The correspondence between such a rule and the cut elimination step in sequent proofs replacing a cut over $A_1 \wedge A_2$ with two cuts over the sub-formulae A_1 and A_2 is quite apparent. This corresponds exactly to a "principal-principal" cut case, called also "key case".

Also the ES reduction rules are modeled on cut elimination steps. For instance, the $Var1$ reduction rule corresponds to the elimination of the last cut in a sequent proof coming from an axiom (see Section) and a similar correspondence can be established for all the other rules of ES. These reduction steps correspond to the so-called "non key-cases". This fact is completely reflected in the full proof of subject reduction given in [10].

This shows that cut elimination provides an enlightening rational reconstruction of computation mechanisms, such as pattern matching and explicit substitution, which are at the heart of the implementation of functional languages.

We use $\longrightarrow_{TPC_{ES}}$ to denote the reduction relation generated by the system TPC_{ES}. The notations $\longrightarrow^{+}{}_{TPC_{ES}}$ and $\longrightarrow^{*}{}_{TPC_{ES}}$ are used to denote, respectively, the transitive and reflexive-transitive closures of $\longrightarrow_{TPC_{ES}}$.

It is worth noticing that if $M \longrightarrow_{TPC_{ES}} N$, then $FV(N) \subseteq FV(M)$; also, ES is locally confluent and terminating so that ES-normal forms are unique. As expected, the calculus enjoys the following property:

Theorem 2.2 (**TPC_{ES} Subject-reduction**) If $\Gamma \triangleright T : D$ and $T \longrightarrow_{TPC_{ES}} T'$, then $\Gamma \triangleright T' : D$.

Proof. By induction on T, using some lemmas that are detailed in [10]. ▪

3. Encoding Functions into the TPC_{ES}-Calculus

This section shows how to encode functions into the TPC_{ES}-calculus. More precisely, we first show that simply typed λ-calculus can be expressed within the TPC_{ES}-calculus in a very simple way. Secondly, we show how to add recursive types (as lists and trees) in order to express programs such as the `merge` example in Section . In both cases, it turns out that conditions on free communication variables used in typing rules $\rightarrow left$, *sub*, *let* and *app* do not limit the expressive power of the TPC_{ES}-calculus.

3.1. Encoding Simply Typed λ-Calculus into TPC_{ES}-Calculus

The simply typed lambda calculus can be immediately translated into the typed pattern calculus. The introduction rules/constructs are already there, so they have a trivial translation. Following the usual translation of natural deduction proofs into sequent proofs, the elimination rules/constructs are translated by the corresponding left rules followed by a let rule.

Let M be a simply typed term. We denote by M^* the translation of M in the pattern calculus, which is recursively defined by:

$$
\begin{array}{l}
x^* =_{def} x \\
(\lambda x\!:\!A.M)^* =_{def} \lambda x\!:\!A.M^* \\
\mathtt{inl}(M)^* =_{def} \mathtt{inl}(M^*) \\
\mathtt{inr}(M)^* =_{def} \mathtt{inr}(M^*) \\
(\pi_i(M))^* =_{def} \mathtt{let}\ M^*\ \mathtt{be}\ \langle x_1, x_2\rangle\!:\!A\times B\ \mathtt{in}\ x_i \\
(MN)^* =_{def} \\
\qquad\qquad \mathtt{let}\ M^*\ \mathtt{be}\ \sharp z : A \to B\ \mathtt{in}\ (z\ \mathtt{of}\ N^*\ \mathtt{is}\ y\!:\!B\ \mathtt{in}\ y) \\
(\text{case } L \text{ of } x\!:\!A.M \mid y\!:\!B.N)^* =_{def} \\
\qquad\qquad \mathtt{let}\ L^*\ \mathtt{be}\ (x \mid_\xi y)\!:\!A+B\ \mathtt{in}\ [M^* \mid_\xi N^*]
\end{array}
$$

where x_1, x_2, y, z, ξ are fresh variables. It is easy to see that the translation is type preserving (since it mirrors a translation of proofs!) and that $(M\{x \leftarrow L\})^* = M^*\{x \leftarrow L^*\}$.

The following lemma justifies the conditions on free communication variables used in typing rules $\to left$, sub, let and app.

Lemma 3.1 Let M be a simply typed term. Then M^* contains no free communication variables.

Indeed, simply typed λ-calculus can be encoded in TPC_{ES}:

Theorem 3.2 (TPC_{ES} simulates β) Let M and N be two simply typed terms. If $M \longrightarrow_\beta N$, then we have that $M^* \longrightarrow^+_{TPC_{ES}} N^*$.

3.2. Adding Recursive Types to TPC_{ES}

A full treatment of the extension of the TPC_{ES} calculus with recursive types, patterns and terms is out of the scope of this extended abstract, but we give here a brief hint of the flavor of such an extension because it will allow us to show how to encode in the TPC_{ES} calculus the example of Section . We refer the interested reader to [10] for further details about technical points concerning this section.

We use the standard fold/unfold [19] technique to encode recursive terms. We also add to the syntax, as is classically done, recursive types and recursive abstractions (denoted by $\mu f.M$) together with the following operational rule for recursion

$$\mu f.M \longrightarrow M[f/\mu f.M]$$

New reduction rules to perform pattern matching between recursive terms and recursive patterns are also added to the reduction system.

All this will allow, for example, to define a particular recursive type to *encode* the type $listA$, particular recursive terms to *encode* the constructors nil and $c(M, L)$ (to denote, respectively, an empty and a non-empty list), and a particular recursive pattern to *encode* the sum pattern $(nil \mid_\xi c(P, Q))$ (used to denote a pattern of recursive type $listA$). Following [10, 27], the following expected reductions sequences can be derived in the extended system:

$$
\begin{array}{l}
\mathtt{let}\ nil\ \mathtt{be}\ (nil \mid_\xi c(P,Q))\ \mathtt{in}\ N \longrightarrow^* N[\xi/\mathtt{L}] \\
\mathtt{let}\ c(M,L)\ \mathtt{be}\ (nil \mid_\xi c(P,Q))\ \mathtt{in}\ N \longrightarrow^* \\
\qquad\qquad \mathtt{let}\ \langle M, L\rangle\ \mathtt{be}\ \langle P, Q\rangle\ \mathtt{in}\ N[\xi/\mathtt{R}]
\end{array}
$$

With this syntactic sugar, we proceed to express the pattern-matching program `merge` given in the introduction. For more clarity, we write $F =_{def} M\{f \leftarrow F\}$ instead of $F =_{def} \mu f.M$ and also we omit types annotations on lambda abstraction.

$$
\begin{array}{l}
\mathtt{merge} =_{def} \\
\lambda\langle @(z_1, ((nil \mid_\xi c(x_1, l_1)))), @(z_2, ((nil \mid_\zeta c(x_2, l_2))))\rangle. \\
\quad [\ [nil \mid_\zeta z_2] \mid_\xi [z_1 \mid_\zeta c(x_1, c(x_2, merge\,\langle l_1, l_2\rangle))]\ ]
\end{array}
$$

This program says that `merge` is a function taking two arguments of type $listA$, each one having the form either nil or $c(\_, \_)$. In order to distinguish the form of the first (resp. second) argument, the communication variable ξ (resp. ζ) is used. The first (resp. second) argument is also called z_1 (resp. z_2) via a layered pattern. The body of the function is expressed by the term

$$[\ [nil \mid_\zeta z_2] \mid_\xi [z_1 \mid_\zeta c(x_1, c(x_2, merge\,\langle l_1, l_2\rangle))]\ ]$$

and considers, as expected, four possible cases, corresponding to four possible instantiations of the sum variables: if the argument is $\langle nil, nil\rangle$, corresponding to $\xi = \mathtt{L}, \zeta = \mathtt{L}$, then the result is nil, if the argument is $\langle nil, c(x_2, l_2)\rangle$, corresponding to $\xi = \mathtt{L}, \zeta = \mathtt{R}$, then the result is z_2, if the argument is $\langle c(x_1, l_1), nil\rangle$, corresponding to $\xi = \mathtt{R}, \zeta = \mathtt{L}$, the result is z_1, and finally if the argument is $\langle c(x_1, l_1), c(x_2, l_2)\rangle$, corresponding to $\xi = \mathtt{R}, \zeta = \mathtt{R}$, then the result is $c(x_1, c(x_2, \mathtt{merge}\langle l_1, l_2\rangle))$.
The closed term `merge` type-checks since the judgment $\rhd\ \mathtt{merge} : listA \times listA \to listA$ is derivable in the extended typing system including recursion [10].

The function `merge` has the expected operational behavior using the new reduction rules recently introduced; indeed, one can verify that:

$$
\begin{array}{lll}
\mathtt{merge}\langle nil, nil\rangle & \longrightarrow^* & nil \\
\mathtt{merge}\langle nil, c(a,l)\rangle & \longrightarrow^* & c(a,l) \\
\mathtt{merge}\langle c(a,l), nil\rangle & \longrightarrow^* & c(a,l) \\
\mathtt{merge}\langle c(a,nil), c(b,nil)\rangle & \longrightarrow^* & c(a, c(b, nil))
\end{array}
$$

where $\mathtt{merge}\langle x, y\rangle$ abbreviates the term $\mathtt{let\ merge\ be}\ \sharp z\ \mathtt{in}\ z\ \mathtt{of}\ \langle x, y\rangle\ \mathtt{is}\ w\ \mathtt{in}\ w$ exactly as in Section .

Other interesting examples using lists can be found in [27]. Note however that in the rest of the paper we will keep on working with a calculus with no recursion, otherwise properties such as strong normalization would not be pertinent.

4. The Calculus TPC

We remark that, in a pattern calculus, both pattern matching and substitutions can be either at the meta-level or in the language itself. More precisely, if both operations are in the meta-language we get, for example, the calculus in [27], if both are in the language itself we get the TPC_{ES} calculus, and, finally, if pattern matching is internalized while substitution is kept as a meta-level operation we get the TPC calculus that we introduce in this section. It is interesting to study here such a calculus because proving some properties of TPC will help us to prove some properties of TPC_{ES} (i.e. confluence and strong normalization).

The (raw) terms of the TPC-calculus are (raw) TPC_{ES}-terms without explicit substitutions. The typing rules of TPC are those for TPC_{ES} except sub, $sub_{\mathtt{L}}$ and $sub_{\mathtt{R}}$. To avoid confusion between typing judgments in TPC and TPC_{ES}, when necessary we will write $\Gamma \;\triangleright_{TPC} M:A$ (resp. $\Gamma \;\triangleright_{TPC_{ES}} M:A$).

The reduction rules of the TPC-calculus are the same as those of $TPC_{ES} \setminus$ ES but explicit substitution is replaced by a meta-level substitution as Figure 3 shows.

In what follows, we use $M\theta$ to denote $M\{y \leftarrow N\}$, or $M\{\xi \leftarrow \mathtt{K}\}$, where $\mathtt{K}$ is $\mathtt{L}$ or $\mathtt{R}$). Note that the notation $M[\theta]$ introduced in Section and the notation $M\theta$ introduced here have different meanings (explicit substitution versus meta-level substitution).

Theorem 4.1 (TPC-Subject Reduction) If $\Gamma \triangleright M:A$ and $M \longrightarrow_{TPC} N$, then $\Gamma \triangleright N:A$.

Proof. The property is a consequence of the subject reduction property for TPC_{ES} (Theorem 2.2) and the fact that ES implements meta-level substitutions. ∎

Theorem 4.2 (Confluence of TPC) $\longrightarrow_{TPC}$ is confluent on raw terms.

Proof. By the Tait and Martin-Löf [4] technique, which relates the reduction relation $\longrightarrow_{TPC}$ to a parallel reduction relation $\gg$. ∎

4.1. Strong Normalization for TPC

In this section we show strong normalization for TPC adapting a proof technique based on stability [28] to the case of pattern matching. We may omit types from terms to make easier the notation but no formal erasure takes place.

We first define the notion of stability for well-typed terms using the notation MN, exactly as in Section , as an abbreviation of the following term:

$$MN =_{def} \mathtt{let}\ M\ \mathtt{be}\ \sharp z : B \to C\ \mathtt{in}\ z\ \mathtt{of}\ N\ \mathtt{is}\ y : C\ \mathtt{in}\ y$$

where z, y are fresh variables, M is of type $B \to C$ and N of type B.

Definition 4.1 (Stable Terms) A term M of type A is defined to be *stable* as follows:

- If A is an atomic type, M is stable iff it is strongly normalizing.
- If $A \equiv A_1 \times A_2$, M is stable iff it is strongly normalizing and whenever M reduces to $\langle M_1, M_2\rangle$, then M_1 and M_2 are both stable.
- If $A \equiv A_1 + A_2$, M is stable iff it is strongly normalizing and whenever M reduces to $\mathtt{inl}_{A_1+A_2}(M')$ or to $\mathtt{inr}_{A_1+A_2}(M')$, then M' is stable.
- If $A \equiv A_1 \to A_2$, M is stable iff for every stable term N of type A_1, MN is stable.

The goal of this section is to show that every *typed* TPC-term is strongly normalizing. For that, as traditionally done, we first show that every stable term is strongly normalizing (Lemma 4.3), and that every typed term is stable (Lemma 4.4).

Lemma 4.3 (1) Every stable term M of type A is strongly normalizing and (2) A term $xK_1 \dots K_n$ of type A is stable for arbitrary strongly normalizing terms $K_1 \dots K_n$.

Proof. The two properties can be shown at the same time by induction on the type A. ∎

The following notion is used in the proof of Lemma 4.4.

Definition 4.2 The measure of a term is a function $\mathcal{M}() : \mathcal{T}_{TPC} \to \mathbb{N}_2$, where $\mathbb{N}_2$ is the set of integers greater or equal than 2, defined as follows:

$$\begin{array}{lcl}
\mathcal{M}(x) & = & 2 \\
\mathcal{M}(\mathtt{inl}(M)) = \mathcal{M}(\mathtt{inr}(M)) & = & \mathcal{M}(M) + 1 \\
\mathcal{M}(\langle M, N\rangle) & = & \mathcal{M}(M) + \mathcal{M}(N) + 1 \\
\mathcal{M}(\lambda P.J) & = & \mathcal{P}(P) + \mathcal{M}(J) + 1 \\
\mathcal{M}(\mathtt{let}\ M\ \mathtt{be}\ P\ \mathtt{in}\ N) & = & \mathcal{P}(P) + \mathcal{M}(N) \\
\mathcal{M}([M \mid_T N]) & = & \mathcal{M}(M) + \mathcal{M}(N) \\
\mathcal{M}(T\ \mathtt{of}\ N\ \mathtt{is}\ Q\ \mathtt{in}\ L) & = & \mathcal{M}(T) * \mathcal{M}(N) * \mathcal{P}(Q) + \mathcal{M}(L)
\end{array}$$

where the notation $\mathcal{P}(P)$ indicates the measure of the pattern P, defined as: $\mathcal{P}(x) = \mathcal{P}(\sharp z) = \mathcal{P}(\_) = 2$ and $\mathcal{P}(@(P,Q)) = \mathcal{P}(\langle P, Q\rangle) = \mathcal{P}((P \mid_\xi Q)) = \mathcal{P}(P) + \mathcal{P}(Q) + 1$.

Lemma 4.4 Every typed term is stable.

$$
\begin{array}{lll}
(\lambda P.J)\ \mathtt{of}\ M\ \mathtt{is}\ Q\ \mathtt{in}\ N & \longrightarrow_{Of} & \mathtt{let}\ (\mathtt{let}\ M\ \mathtt{be}\ P\ \mathtt{in}\ J)\ \mathtt{be}\ Q\ \mathtt{in}\ N \\
\mathtt{let}\ \langle M_1, M_2\rangle\ \mathtt{be}\ \langle P_1, P_2\rangle\ \mathtt{in}\ N & \longrightarrow_{Pair} & \mathtt{let}\ M_1\ \mathtt{be}\ P_1\ \mathtt{in}\ (\mathtt{let}\ M_2\ \mathtt{be}\ P_2\ \mathtt{in}\ N) \\
\mathtt{let}\ M\ \mathtt{be}\ @(P_1, P_2)\ \mathtt{in}\ N & \longrightarrow_{Cont} & \mathtt{let}\ M\ \mathtt{be}\ P_1\ \mathtt{in}\ (\mathtt{let}\ M\ \mathtt{be}\ P_2\ \mathtt{in}\ N) \\
\mathtt{let}\ \mathtt{inl}(L)\ \mathtt{be}\ (P \mid_\xi Q)\ \mathtt{in}\ M & \longrightarrow_{Case1} & \mathtt{let}\ L\ \mathtt{be}\ P\ \mathtt{in}\ M\{\xi \leftarrow \mathtt{L}\} \\
\mathtt{let}\ \mathtt{inr}(L)\ \mathtt{be}\ (P \mid_\xi Q)\ \mathtt{in}\ M & \longrightarrow_{Case2} & \mathtt{let}\ L\ \mathtt{be}\ Q\ \mathtt{in}\ M\{\xi \leftarrow \mathtt{R}\} \\
\mathtt{let}\ (\lambda P.M)\ \mathtt{be}\ \sharp z\ \mathtt{in}\ N & \longrightarrow_{Lambda} & N\{z \leftarrow \lambda P.M\} \\
\mathtt{let}\ M\ \mathtt{be}\ x\ \mathtt{in}\ N & \longrightarrow_{Sub} & N\{x \leftarrow M\} \\
\mathtt{let}\ M\ \mathtt{be}\ \_\ \mathtt{in}\ N & \longrightarrow_{Weak} & N \\
[M \mid_{\mathtt{L}} N] & \longrightarrow_{\mathtt{L}} & M \\
[M \mid_{\mathtt{R}} N] & \longrightarrow_{\mathtt{R}} & N
\end{array}
$$

Figure 3. Reduction Rules of TPC

Proof. To prove this property we need a stronger property:

Let M be a term such that all its free (usual) variables are among $\{x_i\}_{i=1...n}$. If $N_1 \dots N_n$ are stable terms and $\theta = \{x_1/N_1, \dots x_n/N_n\}$ is a well-typed substitution, then $M\theta$ is stable.

The theorem follows by taking $N_i = x_i$. The proof proceeds by induction on $\mathcal{M}(M)$ using a trick proposed in [24] to show strong normalization of λ-calculus mixed with a special kind of higher order rewriting rules. ∎

By Lemma 4.4 and Lemma 4.3 we obtain:

Theorem 4.5 (Strong Normalization for TPC) If $\Gamma \rhd_{TPC} M : A$ then M is TPC-strongly normalizing.

5. Fundamental Properties of TPC_{ES}

This section establishes confluence, preservation of TPC-strong normalization, strong normalization of TPC_{ES} for well-typed terms, and reductibility to "values" for closed well-typed terms.

The proof technique used to show confluence is the interpretation method [20], which *maps* reduction sequences of a rewriting relation to sequences of another relation for which we already know confluence. In our case, TPC_{ES}-reduction is simulated by TPC-reduction, so that confluence of TPC_{ES} is a consequence of confluence of TPC. Thus, we get the following result:

Theorem 5.1 (Confluence of TPC_{ES}) The relation TPC_{ES} is confluent.

Proof. By the interpretation method [20], using the fact that TPC is confluent (by Theorem 4.2), the fact that a TPC reduction step can be simulated by a TPC_{ES} reduction sequence and the fact that whenever $a \longrightarrow_{TPC_{ES}} b$, then $\mathrm{ES}(a) \longrightarrow^*_{TPC} \mathrm{ES}(b)$, where $\mathrm{ES}(a)$ is the normal form of a w.r.t the system ES. ∎

In contrast to strong normalization, which is a property usually proved for *well-typed* terms, preservation of strong normalization is a property on *raw* terms, and it is, in some sense, a property that guarantees the correctness of the reduction relation with explicit substitutions with respect to the corresponding reduction relation with meta-level substitutions.

Definition 5.1 (S Preserves R-strong normalization) Let $(R, \mathcal{T}_R)$ and $(S, \mathcal{T}_S)$ be two reduction systems such that $\mathcal{T}_R \subseteq \mathcal{T}_S$. We say that S preserves R-strong normalization (or that PSN for S holds) if every term in $\mathcal{T}_R$ which is R-strongly normalizing is also S-strongly normalizing.

To show that $\longrightarrow_{TPC_{ES}}$ preserves $\longrightarrow_{TPC}$ -strong normalization of raw TPC-terms, we introduce the following definition, where the notation SN_R is used to denote the set of all terms that are R-strongly normalizing and the notation $N \subseteq M$ is used as a shorthand for "N is a subterm of M".

Definition 5.2 The set $\mathcal{F}$ is defined as $\{M \mid M \in \mathcal{T}_{TPC_{ES}}$ and $\forall N \subseteq M\ \ \mathrm{ES}(N) \in SN_{TPC}\}$.

Lemma 5.2 $\mathcal{F} \subseteq SN_{TPC_{ES}}$.

Proof. We use the technique proposed by Bloo and Geuvers in [6, 7]. We first define a signature Σ together with a precedence $\succ$ on its symbols so that $>_{rpo}$ is well-founded on the set $\mathcal{T}_\Sigma$ of variable-free terms over the signature Σ [14, 25]. We then define a translation $\mathcal{R}()$ from $\mathcal{F}$ to $\mathcal{T}_\Sigma$. Since it can be shown that $\mathcal{F}$ is closed under TPC_{ES} reduction, then we get, by induction on the structure of terms, that, if $M \in \mathcal{F}$ and $M \longrightarrow_{TPC_{ES}} N$, then $\mathcal{R}()$ is also defined on N and $\mathcal{R}(M) >_{rpo} \mathcal{R}(N)$ (see [10] for the full proof). This allows us to conclude that any term in $\mathcal{F}$ is also TPC_{ES}-strongly normalizing. ∎

Theorem 5.3 (PSN for TPC_{ES}) $\longrightarrow_{TPC_{ES}}$ preserves $\longrightarrow_{TPC}$ -strong normalization of TPC raw terms.

Proof. Take $M \in SN_{TPC}$. Then $M \in \mathcal{F}$. Hence, by Lemma 5.2. $M \in SN_{TPC_{ES}}$. ▪

Now, since one can show that any term typable in TPC_{ES} belongs to $\mathcal{F}$, then we can conclude by Lemma 5.2 that

Theorem 5.4 (Strong Normalization for TPC_{ES}) If $\Gamma \triangleright_{TPC_{ES}} M : A$, then $M \in SN_{TPC_{ES}}$.

Even if explicit substitutions can always be eliminated, it is not true, in general, that the TPC_{ES} normal form of a term does not contain the let constructor. Indeed, the term $\texttt{let}\ x\ \texttt{be}\ \langle y, v\rangle\ \texttt{in}\ z$ contains a let constructor and is in TPC_{ES} normal form. This is quite natural since pattern matching cannot be performed on terms that are not sufficiently specified or instantiated.

Say that a closed term is a *Value* when it is in TPC_{ES} normal form and it has one of the following forms: $\langle M, M\rangle$, $\texttt{inl}_A(M)$,$\texttt{inr}_A(M)$, $\lambda P : A.M$. A natural question arises: is it possible to evaluate any given (functional) *program* (modeled by a TPC_{ES} *closed* term), to a *result* (modeled by a value)? The answer to such a question is positive. That is, we have the following result:

Theorem 5.5 (Reduction to Values) Let M be a *closed* and well-typed term of TPC_{ES}. Then M reduces to a value.

Proof. Since M is a typed term, then it is TPC_{ES}-strongly normalizing by Theorem 5.4. Then the proof proceeds by induction on the maximal length of a TPC_{ES}-reduction sequence starting at M. ▪

6. Conclusion

This paper gives the first rational reconstruction of pattern matching via cut elimination. The main ideas and results can be summarized by the following points:

- A typed pattern calculus TPC_{ES} is proposed, where both typing and reduction are modeled on sequent calculus in the spirit of the Curry-Howard isomorphism.
- Pattern matching and substitution computations are *explicitly* coded in TPC_{ES} and correspond exactly to cut elimination in sequent proofs.
- The calculus enjoys those fundamental properties required by any well-behaved calculus, such as: subject reduction, confluence, preservation of strong normalization w.r.t. the system TPC, strong normalization and reduction to values for closed terms.

Our calculus is a major step forward with respect to the other proposals that have been made these last years in a similar direction. Abramsky [2] gives a term assignment for the intuitionistic sequent proofs, but the terms are the same as those that arise from the term assignment to natural deduction proofs. His interpretation of the sequent proof rules gives an alternative, but equivalent, set of type-checking rules. Gallier [17] gives a novel term assignment to sequent proofs and describes the cut-elimination rules on it, but his syntax does not allow to build nested patterns. He just suggests some reduction rules by interpreting cut elimination in his calculus, but no full reduction system is studied. Lafont [29] proposes a computational interpretation of the sequent proofs for intuitionistic propositional logic under the name of *clausal calculus*. He also uses sequent left rules to build nested patterns but only some reduction rules are suggested in the spirit of cut elimination, while the explicit substitution paradigm is not incorporated at all. Herbelin [21] proposes a calculus which is also inspired by the Curry Howard isomorphism. His calculus does not have abstraction on patterns but only on variables, and reductions of terms correspond exactly to cut-elimination rules. However, β-reduction cannot be simulated by this calculus because substitutions cannot traverse redexes. This is in the same spirit of the λ_χ-calculus with explicit substitutions in [34]. Another calculus in the spirit of the Curry Howard isomorphism is presented in [39]. There, Wadler gives a computational interpretation of the same logical fragment that we study in this paper, via a formalism that only allows abstractions on non-nested patterns. Fundamental properties such as confluence and strong normalization are not studied. In [38], Vestergaard studies the correspondence between λ-calculi with explicit substitutions (without patterns) in de Bruijn notation and intuitionistic logic in Gentzen style. He has to deal with many notions that only appear in a setting with de Bruijn indices, so that the presentation of his cut elimination rules stays very technical. We prefer to explain our approach in a named-variable notation because it makes some essential properties of explicit substitutions more apparent by abstracting out the details of renaming and updating. Another interpretation of explicit substitutions in terms of cut elimination is given by Di Cosmo and Kesner in [16]. There, it is shown the existing correspondence between normalization in calculi with explicit substitution and cut elimination in linear logic, via proof nets. Another correspondence between a calculus with explicit substitution and intuitionistic logic in natural deduction style is established in [35]. These two last approaches study the logical meaning of a given reduction system with explicit substitutions rather than the computational meaning of a given logical cut elimination process. Finally, it is worth noticing that the TPC_{ES}-calculus is not a particular case of the higher order calculi with explicit substitutions proposed in [8], which do not allow to make abstractions on patterns, neither to have different forms of explicit substitutions.

A subject of future investigation might be the study of the operational semantics of TPC_{ES} by means of *lazy* and *eager* evaluators, corresponding to, respectively, call-by-name and call-by-value computation strategies. This would give richer strategies than the ones for calculi with meta-level operations [27], because explicit pattern matching and substitutions allow to specify finer control over reduction. Also, we think that the calculus can be extended with polymorphic types, patterns and terms as it is done, for instance, in [12]. Since the calculus TPC_{ES} provides a computational interpretation of minimal logic sequent calculus, and there is a deep connection between $S4$ modal logic and LISP's `eval` and `quote` primitives in functional programming [13, 18], it would be interesting to study whether our formalism might be extended to that logic.

Acknowledgments

The second author would like to thank Val Tannen for valuable preliminary discussions concerning cut elimination and patterns.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *JFP*, 4(1):375–416, 1991.

[2] S. Abramsky. Computational interpretations of linear logic. *TCS*, 111:3–57, 1993.

[3] The Alfa WWW page. http://www.cs.chalmers.se/ hallgren/Alfa/.

[4] H. Barendregt. *The Lambda Calculus; Its syntax and Semantics (revised edition)*. North Holland, 1984.

[5] Z. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. $\lambda\upsilon$, a calculus of explicit substitutions which preserves strong normalisation. *JFP*, 6(5):699–722, 1996.

[6] R. Bloo. *Preservation of Termination for Explicit Substitution*. PhD thesis, Eindhoven University of Technology, 1997.

[7] R. Bloo and H. Geuvers. Explicit substitution: on the edge of strong normalization. *TCS*, 211(1-2):375–395, 1999.

[8] R. Bloo and K. Rose. Combinatory reduction systems with explicit substitution that preserve strong normalisation. In *RTA*, LNCS 1103, 1996.

[9] R. Burstall, D. MacQueen, and D. Sanella. Hope: An experimental applicative language. In *Proc. of the LISP Conference*, pages 136–143, Stanford University, Computer Science Department, July 1980.

[10] S. Cerrito and D. Kesner. Pattern matching as cut elimination. Technical report 1211, LRI, Université Paris-Sud, France, 1999.

[11] The Coq WWW page. http://pauillac.inria.fr/coq/index.html.

[12] P. Courtieu. *Extension au second ordre du λ-calcul typé avec motifs*. Mémoire de DEA, Université de Paris XI, 1996.

[13] R. Davies and F. Pfenning. A modal analysis of staged computation. In *23rd POPL*, 1996.

[14] N. Dershowitz. Orderings for term rewriting systems. *TCS*, 17(3):279–301, 1982.

[15] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–309. North Holland, 1990.

[16] R. Di Cosmo and D. Kesner. Strong normalization of explicit substitutions via cut elimination in proof nets. In *LICS*, 1997.

[17] J. Gallier. Constructive logics part I: A tutorial on proof systems and typed λ-calculi. *TCS*, 110, 1993.

[18] J. Goubault-Larrecq. On computational interpretations of the modal logic S4: Cut elimination, 1996. Available as http://theory.doc.ic.ac.uk:80/theory/guests/GoubaultJ/eq1.dvi.gz.

[19] C. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing Series. The MIT Press, 1992.

[20] T. Hardin. *Résultats de confluence pour les règles fortes de la logique combinatoire catégorique et liens avec les lambda-calculs*. Thèse de doctorat, Univ. de Paris VII, 1987.

[21] H. Herbelin. A λ-calculus structure isomorphic to sequent calculus structure. In *CSL*, LNCS 845, 1994.

[22] The HOL WWW page. http://www.dcs.gla.ac.uk/ tfm/fmt/hol.html.

[23] P. Hudak, S. Peyton-Jones, and P. W. (editors). Report on the programming language haskell, a non-strict, purely functional language (version 1.2). Sigplan Notices, 1992.

[24] J.-P. Jouannaud and M. Okada. Abstract data type systems. *TCS*, 173(2):349–391, feb 1997.

[25] S. Kamin and J.-J. Lévy. Attempts for generalizing the RPO. University of Illinois, 1980.

[26] D. Kesner. Confluence of extensional and non-extensional lambda-calculi with explicit substitutions. *TCS*. To appear.

[27] D. Kesner, L. Puel, and V. Tannen. A Typed Pattern Calculus. *I&C*, 124(1):32–61, 1996.

[28] J.-L. Krivine. *Lambda-calcul, types et Modèles*. Masson, 1990.

[29] Y. Lafont. Functional programming and linear logic. Lecture Notes for the Summer School on Functional Programming and Constructive Logic, 1989.

[30] The LEGO WWW page. http://www.dcs.ed.ac.uk/home/lego/.

[31] X. Leroy et al. The Caml Light system, release 0.6. Available on `ftp.inria.fr`, 1993.

[32] P.-A. Mellies. Typed λ-calculi with explicit substitutions may not terminate. In *TLCA*, LNCS 902, 1995.

[33] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, 1990.

[34] C. Muñoz. Confluence and preservation of strong normalisation in an explicit substitutions calculus (extended abstract). In *LICS*, 1996.

[35] B. Pagano. Une déduction naturelle explicite. In *Proc. of the 1st Westapp*, 1998.

[36] The PVS WWW page. http://www.csl.sri.com/pvs/overwiew.html.

[37] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *FPLCA*, LNCS 201, 1985.

[38] R. Vestergaard. The cut rule and explicit substitutions, 1998. Draft.

[39] P. Walder. A curry-howard isomorphism for sequent calculus, 1993. Draft.