Optimality for Dynamic Patterns

Extended Abstract

Thibaut Balabonski

Preuves, Programmes et Systèmes
PPS - CNRS and Université Paris-Diderot, France
thibaut.balabonski@pps.jussieu.fr

Abstract

Evaluation of a weak calculus featuring expressive pattern matching mechanisms is investigated by means of the construction of an efficient model of sharing. The sharing theory and its graph implementation are based on a labelling system derived from an analysis of causality relations between evaluation steps. The labelled calculus enjoys properties of confluence and finite developments, and is also used for proving correctness and optimality of a whole set of reduction strategies.

Categories and Subject Descriptors F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Lambda Calculus and Related Systems; D.3.3 [*Programming Languages*]: Language Constructs and Features—Patterns; I.1.3 [*Symbolic and Algebraic Manipulation*]: Languages and Systems—Evaluation Strategies

General Terms Theory, Languages

Keywords Pattern matching, Dynamic patterns, Sharing, Optimality, Labelled calculi

1. Introduction

The motivation of this work is to go toward an efficient implementation model for functional programming languages featuring expressive pattern matching facilities. As a first step, this paper studies sharing and optimality in a calculus with patterns.

Pattern matching can be simply understood as a basic mechanism used to define functions by cases on the structure of the argument; it is one of the main aspects revealing the interest of the functional paradigm for the working programmer. Unfortunately usual mechanisms suffer from some lack of genericity explained in the examples below.

Example 1.
Consider a structure of binary tree: a tree is either a single data or a node with two subtrees, which could be written in ML-style as follows.

```
type 'a tree =
   | Data of 'a
   | Node of 'a tree * 'a tree
```

A function `upd` *updating all data in binary trees can be easily written by recursion on the inductive structure of trees, using pattern matching.*

```
let rec upd f t = match t with
  | Data a   -> Data (f a)
  | Node l r -> Node (upd l) (upd f r)
```

In Example 1, when the function `upd` is applied to arguments f and t, the latter is compared to the shape `Data a`, called a **pattern** (in the whole paper, bold font is used for definitions and technical vocabulary). In case of **success** (the pattern and the argument match, *i.e.* $t = \texttt{Data}\ u$ for some u), the **matching variable** `a` captures the substructure u of the argument t and then the updated data $\texttt{Data}\ (f(u))$ is returned. If the first comparison **fails** (pattern and argument don't match), then the **alternative case** `Node l r` is tested.

The code of `upd` names explicitly all the constructors which will be met during evaluation, that is, `Data` and `Node`. Thus, any minor variation on the data structure (trees of other or arbitrary arities, lists, heterogeneous data...) requires a new definition of the `upd` function.

A first solution for this problem is given by **path polymorphism** [16, 18], which allows a function to go recursively through any data structure and act on some fixed base cases (like `Data a` here). Path polymorphism can be achieved by a universal –and very simple– classification of structures: each one is either atomic, or compound. Atomic structures (called **atoms**), such as constructors, are inert. Compound structures (called **compounds**), such as (partially) applied constructors, lead to recursion.

Example 2.
The code below shows how a path polymorphic version of `upd` *could look like.*

```
let rec upd f t = match t with
  | Data a -> Data (f a)            //Base
  | x y    -> (upd f x) (upd f y)   //Compound
  | z      -> z                     //Atom
```

Example of evaluation of the path polymorphic `upd` *(brackets are used as parentheses to separate function calls, and* `++` *is the successor function):*

```
          upd ++ (Node  (Data 1) (Data 2))
      [upd ++ (Node (Data 1))]  [upd ++ (Data 2)]
[upd ++ Node] [upd ++ (Data 1)] [upd ++ (Data 2)]
        Node          (Data 2)          (Data 3)
```

While `Node` *alone is an atom, terms like* `Node (Data 1)` *and* `Node (Data 1) (Data 2)` *are compounds.*

PPDP'10, July 26–28, 2010, Hagenberg, Austria.

The decomposition in compounds and atoms can be encoded in usual languages with the use of an explicit constructor for compounds. Note that this solution gives up typing. A more subtle approach is in Generic Haskell [13] where reasoning can be done by case on the type of the data structure.

As shown with `upd`, some constructors (as `Node`) can be treated automatically by pattern polymorphism, but some others (as `Data`) have still to be explicitly written in the code, thus limiting polymorphism: if a new structure is given in which relevant elements are flagged by any means different from the specific constructor `Data`, then a new `upd` function has to be defined.

The second solution, solving the latter problem, is given by **pattern polymorphism** [17, 18], which allows to parametrize a pattern by an arbitrary term, thus allowing different instantiations of the same pattern. The simpler case is when the pattern parameter is just a constructor, the more general and interesting one is when the pattern parameter is a *function constructing a pattern*. Such a function needs to be evaluated before pattern matching is performed, thus patterns become **dynamic**. This is completely out of the scope of usual programming languages.

Example 3.
A new version of `upd` *using both path and pattern polymorphisms would as follows take an additional parameter* `c` *to build its pattern for the base case:*

```
let rec upd c f t = match t with
  | {a}    c a  ->  c (f a)
  | {x,y}  x y  ->  (upd c f x) (upd c f y)
  | {z}    z    ->  z
```

where each case contains a list of names to discriminate between matching variables and pattern parameters.

Remark that partial application of the `upd` function of Example 3 to the constructor `Data` yield exactly the version of `upd` of Example 2, but now other constructors or structures can be used to describe the elements to be updated!

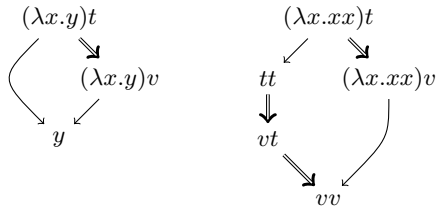
Dynamic patterns are really a key to expressive pattern matching paradigms, but they are incompatible with usual implementations. Indeed pattern matching definitions are normally compiled into series of low-level tests, organized in what is generically called a **matching automaton**. Since a naive matching automaton is likely to introduce a lot of redundant tests and other inefficiencies, sequences of patterns are analyzed in order to get optimized control structures which minimize the average cost of pattern matching and share redundant tests. See for instance [11, 15, 27]. However, dynamic patterns are by nature undefined at compile-time and make this kind of static analysis impossible: new mechanisms have to be invented to recover some of the lost optimizations.

The aim of this work is to conciliate dynamic patterns and efficient evaluation by introducing run-time sharing mechanisms that minimize redundancy of matching operations.

Whereas both aspects are often separated, the study of sharing provided by this paper includes pattern matching in the kernel of functional programming languages (as can be found in higher-order rewriting). The framework used to model higher-order functions, data structures and pattern matching with dynamic patterns is the *Pure Pattern Calculus* (*PPC*) of Jay and Kesner [17–20]. The formalism encompasses λ-calculus and usual pattern matching, allowing to apply any relevant result obtained here to the standard cases. Furthermore, this framework is even richer since it realizes both path and pattern polymorphisms, which is why it is preferred here to other pattern matching calculi such as the λ-calculus with patterns [21, 23] or the rewriting calculus [8].

Taking advantage of the unified formalism of *PPC*, the discussion on efficiency led here concerns pattern matching as well as function calls. The paper thus proceeds by extending to the more general *PPC* some concepts of λ-calculus that are presented below.

A **reduction strategy** is the choice of an order for reduction steps of a term. Suppose the term t evaluates to the value v. The figure below illustrates how the choice of a strategy can have an impact on the number of reduction steps, and then on the evaluation cost. In both pictures the left path is *call-by-name* while the right path is *call-by-value*.

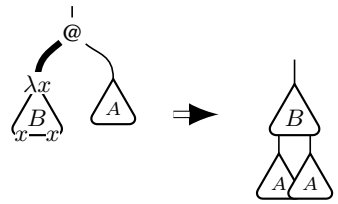


In the left picture, the difference comes from the presence of some useless fragment in the program, whereas in the right picture there is a possibility of duplication of a non-evaluated program. An efficient reduction strategy has to cope with these two pitfalls.

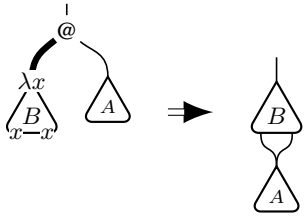
Whereas useless evaluation can be avoided (by so-called needed reduction), it is known that an evaluation order alone is not enough to prevent duplication [24]. Thus an efficient implementation comes from the combination of a reduction strategy with a mechanism of **sharing** of duplicated terms. The idea is to make sure that some parts of a program which are *logically* duplicated (in the term representation of the program) remain *physically* single pieces (in the memory of the evaluator).

The use of sharing leads from classical term representations of programs (say, as higher-order rewriting systems [32]) to graph representations [21], where duplication of a whole subterm may be replaced by the copy of a single pointer, as showed below.

Pictures will take the form of syntactic trees or graphs, top-down oriented. Application is denoted by the binary node @, and redexes are marked with bold lines (**red***ucible* **ex***pression*, or **redex**, designs a place where an evaluation step can take place). For instance in the following picture, an abstraction $\lambda x.B$ is applied to an argument A. The function body B contains two occurrences of the formal parameter x, and the argument A is thus logically duplicated, representing **call-by-name**:

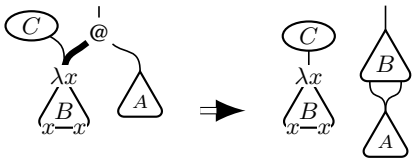


So-called **lazy** evaluation (next figure) prevents this duplication: A stays physically unique, with two pointers to its location.

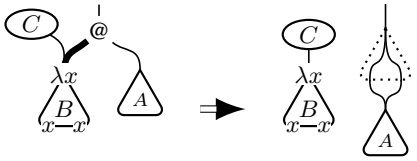


The reader should keep in mind that the term laziness is to be taken in literal sense: if there is something to do (as a duplication), just wait for some better reason, which often appears quite fast. In par-

ticular a shared function has to be copied prior to any instantiation, as shown in the picture below.

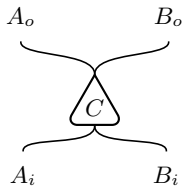


A further step in the hierarchy of lazinesses is called **fully lazy** evaluation, and performs only a partial duplication of the body of the function, namely the parts that depends on the formal parameter [34]. In the next picture, the dotted zone represents the parts whose copy is (at least momentarily) saved by full laziness.



The graphs generated by this kind of reduction are directed acyclic graphs (DAGs) in which there is at most one arrow out of each node. An important consequence is that in this model, only complete subterms are shared, and the graphs are precise representations of what is present in the memory of the evaluator. Each node is a memory location, each edge is a pointer, and nothing more is needed for evaluation. It can also be noticed that any such graph can be easily *unfolded* (or **read-back**) into the term it represents.

Further levels of laziness [14, 24] lose these interesting properties. Indeed they need to share subcontexts (or open terms) instead of complete subterms [1, 31]: something which is shared at some point can be unshared later.



The problem here is to match correctly the inputs with the outputs. For correct evaluation the right associations have to be remembered, which needs to introduce some additional control structures. The possibly huge bookkeeping work induced by these additional structures [1] prevents optimal β-reduction from being the unquestionable optimal choice of implementation.

As a consequence, any really efficient model of sharing has to strike a balance between the effective amount of sharing and the additional implementation cost. The choice here is full laziness, as it is the finest known level of sharing that shares only complete subterms and then avoids the explosion of hidden costs that characterizes optimality. Moreover, this presumed efficiency of full laziness is effective: see for instance the implementation for λ-calculus proposed in [30].

All the reasoning on graph representations appearing in this paper is done through enriched terms representing graphs. To achieve this, terms are decorated with labels representing memory locations and pointers (see Section 4). Since memory locations determine what is shared or copied, the interesting point is the way in which the labels are designed (Section 3).

As in Lévy's optimality theory [25], labels are meant to characterize *equivalent* subterms (originally equivalent redexes) that should be shared (in other words *that should not have been separated*). Informally, being equivalent means to have a common origin, followed by equivalent evolutions (the whole being called **history**).

The kernel of this work is a representation of the history of terms by labels derived from an analysis of causality relations between evaluation events.

In the simple functional case a redex appears wherever an abstraction is applied to an argument, and the only relevant history in this case can be recorded locally as in [4, 25]. However, as shown in Example 4 an argument that has to be matched against some pattern is required to be in a suitable form.

Example 4.
Consider for the following program.

```
let id x = x
and rec count = fun
  | Data a    ->  1
  | Node l r  ->  (count l) + (count r)
in
  count (id (Data 0))
```

Function `count` *is called with argument* `id (Data 0)`*, which has to be evaluated to* `Data 0` *before the application can be resolved.*

Thereby a redex also depends on the history of its subterms, at an arbitrary depth. Moreover, notice that Example 4 do not use the new features presented here: this point already holds for usual pattern matching.

These non-local contributions associated with a generic matching algorithm represent a new challenge and a major source of difficulty for this work, especially since pattern matching is allowed to fail. Moreover, in *PPC* this challenge concerns not only the arguments but also the (dynamic) patterns! Difficulty of this point is detailed in Subsection 3.1 (last paragraph) once all needed material is available, whereas novelty is argued in Section 6 (related work).

Main originality of the paper is the advanced treatment of history and contributions, which is still unexplored for the kind of pattern matching presented here, as far as the author is aware.

As done in current implementations for functional languages, weak evaluation is considered here. More precisely, *PPC* will be restricted by constraining evaluation inside the body of an uninstantiated function (which means *partial evaluation*). The choice of this weak version follows an approach by Blanc, Lévy and Maranget and is closely related to fully lazy sharing [4]. Please note that this work considers optimality relative to the weak restriction, which differs from the usual strong theory of β-optimality mentioned above.

Correctness and optimality of a whole class of strategies for the graph implementation of *PPC* proposed in this paper are stated (Correctness & Optimality Corollary 7) by means of an axiomatic approach given by Glauert and Khasidashvili [12]. They provide abstract notions of *equivalent redexes* and *contribution to a redex* using a set of axioms which are sufficient to prove that some strategies turn out to be optimal.

Main difficulty for the user of such a technology is to exhibit concrete notions which are clever enough to satisfy all the axioms defining the abstract notions. The technique proposed here to construct satisfying notions of equivalence and contribution is a direct reuse of the labelling system that defines the graph implementation, which gives the axioms almost for free. This shows how the systematic analysis of history can base various advanced results and constructions on a system.

A second contribution of this work is to show on the example of *PPC* an alternative method to derive optimality of needed strategies in concrete rewriting systems.

Summarizing, the present paper fills the two empty cells of the following table:

λ-calculus	Pure Pattern Calculus [18]
Weak λ-calculus [7]	
Optimality for weak λ-calculus [4]	

That is, a weak version of PPC is proposed, for which an optimality theory is then developed by means of Lévy's labels. This gives a positive answer to a question raised in the conclusion of [23] (originally for λ-calculus with patterns).

Organization in short: Section 2 formally introduces the *Pure Pattern Calculus* and defines its weak restriction. Section 3 analyzes the ways different redexes can contribute to each other, and deduces a confluent labelled pattern calculus enjoying finite developments properties (Confluence Theorem 2 and Finite Developments Theorem 4). The labelled calculus is linked to a graph reduction system in Section 4 (through Preservation of Sharing Theorem 6). Section 5 finally states correctness and optimality of some reduction strategies (Correctness & Optimality Corollary 7). Related work is reviewed in Section 6, before a conclusion is drawn.

By lack of space the technical details cannot be all included in this extended abstract. The interested reader is referred to [3] for further information.

2. First-Class Patterns

2.1 The Pure Pattern Calculus

The *Pure Pattern Calculus* (*PPC*) of Jay and Kesner [18] is a functional framework featuring data structures and pattern matching with path and pattern polymorphisms. One of its attributes is to make patterns first-class citizens: they can be given as arguments or returned as results. They can also be evaluated so that they are called **dynamic patterns**. Moreover, any term is allowed to be used as a pattern *a priori*, the pattern matching operation being responsible for dynamically rejecting *improper* uses. Here is a reminder of the calculus, followed by the definition of its weak restriction.

Syntax. The grammar for terms is associated with its subcategory of **matchable forms**: *stable* terms that are ready for matching.

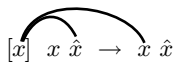
$$\begin{array}{rcll} a,b,p,r,s,t & ::= & x \mid \hat{x} \mid tt \mid [\theta]\, t \to t & \text{Terms} \\ d & ::= & \hat{x} \mid dt & \text{Data structures} \\ m & ::= & d \mid [\theta]\, t \to t & \text{Matchable forms} \end{array}$$

where $x, y, z \in \mathcal{X}$ the set of names, and θ, τ are lists of names.

The term $t_1 t_2$ is called an **application**, and $[\theta]\, p \to b$ a **case**. Letter p indicates a term used as ***p***attern, and b as a function ***b***ody. Letters a, r and s are also used below for ***a***rguments, ***r***edexes and ***s***ubterms. As usual, let $\mathcal{C}[]$ denote a context (term with a hole) and $\mathcal{C}[t]$ the context $\mathcal{C}[]$ where the hole is replaced by t. The natural notion of subterm of t at position $\mathfrak{p}$ is written $t|_{\mathfrak{p}}$.

Note that a name x has two kinds of occurrences: proper occurrences x as **variables** which can be substituted, and occurrences $\hat{x}$ as **matchables** which can't. The matchable $\hat{x}$ is a constant with a name x: it may be used either as a constructor or as a matching variable (which captures what will be substituted for x), depending on its free or bound status. Boldface symbol **c** denotes a ***c***onstructor in the examples.

Variable and matchable bindings. As pictured below, in the term $[\theta]\, p \to b$ the list θ binds matchables in p and variables in b.



This corresponds to the following formal definitions for free variables $fv(t)$ and free matchables $fm(t)$ of a term t. **Free names** of a term t are defined as $fn(t) = fv(t) \cup fm(t)$.

$$\begin{array}{rcl} fv(x) & := & \{x\} \\ fv(\hat{x}) & := & \emptyset \\ fv(t_1 t_2) & := & fv(t_1) \cup fv(t_2) \\ fv([\theta]\, p \to b) & := & fv(p) \cup (fv(b) \setminus \theta) \end{array}$$

$$\begin{array}{rcl} fm(x) & := & \emptyset \\ fm(\hat{x}) & := & \{x\} \\ fm(t_1 t_2) & := & fm(t_1) \cup fm(t_2) \\ fm([\theta]\, p \to b) & := & (fm(p) \setminus \theta) \cup fm(b) \end{array}$$

A natural notion of α-conversion is deduced from these binding rules. For any $x \in \theta$ and y fresh:

$$[\theta]\, p \to b \quad =_\alpha \quad [\theta\{x := y\}]\, p\{\hat{x} := \hat{y}\} \to b\{x := y\}$$

For now on, bound names of any term will be considered all different (and also different from the free names).

Example 5.

$$[y]\, \hat{x}y\hat{y} \to xy\hat{y} =_\alpha [z]\, \hat{x}y\hat{z} \to xz\hat{y}$$

In the pattern $\hat{x}y\hat{y}$ the matchable $\hat{y}$ is bound and is used as a matching variable, whereas $\hat{x}$ is free and seen as a constructor. y is a free variable: an external parameter.

Substitution. Substitution, as in λ-calculus, is a meta-operation defined by equations.

$$\begin{array}{rcll} x^\sigma & := & \sigma_x & x \in dom(\sigma) \\ x^\sigma & := & x & x \notin dom(\sigma) \\ \hat{x}^\sigma & := & \hat{x} & \\ (t_1 t_2)^\sigma & := & t_1^\sigma t_2^\sigma & \\ ([\theta]\, p \to b)^\sigma & := & [\theta]\, p^\sigma \to b^\sigma & \theta \# \sigma \end{array}$$

The condition $\theta\#\sigma$ (read θ **avoids** σ) is here to prevent name capture. $\theta\#\sigma$ stands for $\theta \cap (dom(\sigma) \cup fn(cod(\sigma))) = \emptyset$, where $dom(\sigma)$ (resp. $cod(\sigma)$) denotes the domain (resp. codomain) of the substitution σ. The condition $\theta \cap dom(\sigma) = \emptyset$ ensures that no bound variable is substituted, while $\theta \cap fv(cod(\sigma)) = \emptyset$ (resp. $\theta \cap fm(cod(\sigma)) = \emptyset$) ensures that no variable (resp. matchable) of σ is captured by θ in b^σ (resp. p^σ). For terms, $\theta\#t$ is $\theta \cap fn(t) = \emptyset$.

Pattern matching. The result of the pattern matching operation is called a **match** (meta-variable μ), and is either a substitution in case of successful matching, or the symbol $\perp$ for matching failure. The match of an argument a against a pattern p with matching variables θ is noted $\{a/[\theta]\, p\}$. Its definition is based on the following **compound matching** operation, where equations are to be taken in order:

$$\begin{array}{rcll} \{\!\{a/[\theta]\, \hat{x}\}\!\} & := & \{x \mapsto a\} & x \in \theta \\ \{\!\{\hat{x}/[\theta]\, \hat{x}\}\!\} & := & \{\} & x \notin \theta \\ \{\!\{a_1 a_2/[\theta]\, p_1 p_2\}\!\} & := & \{\!\{a_1/[\theta]\, p_1\}\!\} \uplus \{\!\{a_2/[\theta]\, p_2\}\!\} & \\ & & a_1 a_2 \text{ and } p_1 p_2 \text{ are matchable forms} & \\ \{\!\{a/[\theta]\, p\}\!\} & := & \perp & \\ & & a \text{ and } p \text{ are matchable forms, otherwise} & \\ \{\!\{a/[\theta]\, p\}\!\} & := & \texttt{undefined} & \text{otherwise} \end{array}$$

where the $\uplus$ operator stands for disjoint union of substitutions. Union of matches $\mu_1 \uplus \mu_2$ is $\perp$ if μ_1 or μ_2 is $\perp$ or if the domains of μ_1 and μ_2 overlap. The latter condition ($\sigma_1 \uplus \sigma_2 = \perp$ if $dom(\sigma_1)$ and $dom(\sigma_2)$ overlap) implies that a match cannot succeed if its pattern is not linear, which is a basic requirement for confluence [22]. An undefined result corresponds to the case where the pattern or the argument has still to be evaluated or instantiated.

Example 6.

Let $\mathbf{c}$ be a constructor. The match $\{\!\{\mathbf{c}/[y]\, x\hat{y}\}\!\}$ is undefined since the pattern $x\hat{y}$ starting with a variable is not a matchable form. The match $\{\!\{\mathbf{c}/[y]\, ([z]\, \hat{z} \to z\mathbf{c})\hat{y}\}\!\}$ is also undefined: the pattern is now a preredex (definition below), which is not a matchable form either. The third attempt $\{\!\{\mathbf{c}/[y]\, \hat{y}\mathbf{c}\}\!\}$ is defined as $\bot$, since the atomic argument $\mathbf{c}$ do not match the compound pattern $\hat{y}\mathbf{c}$. An example of successful compound matching is $\{\!\{\mathbf{c}([z]\, \hat{z} \to z\mathbf{c})/[x_1 x_2]\, \hat{x_1}\hat{x_2}\}\!\}$: the argument and the pattern are matchable forms, and each of the bound matchables $\hat{x_1}$ and $\hat{x_2}$ captures a part of the argument to yield the substitution $\{x_1 \mapsto \mathbf{c},\ x_2 \mapsto ([z]\, \hat{z} \to z\mathbf{c})\}$.

Now suppose $\{\!\{a/[\theta]\, p\}\!\} = \sigma$. If $\theta = dom(\sigma)$ then define $\{a/[\theta]\, p\} = \sigma$, else $\{a/[\theta]\, p\} = \bot$. This **check** operation ensures that the pattern p contains all the matchables whose names are bound by θ.

Reduction. As in λ-calculus, reduction is defined by one single rule (called β_m) which is still a meta-operation, performing pattern matching and substitution in one step. Any subterm of the form $r = ([\theta]\, p \to b)a$ is called a **preredex**. If $\{a/[\theta]\, p\}$ is defined then the preredex r is a **redex** and the rule β_m applies. For any term b, define $b^{\bot}$ as some fixed closed normal form $\bot$.

$$([\theta]\, p \to b)a \quad \xrightarrow{r}_{\beta_m} \quad b^{\{a/[\theta]\, p\}}$$

The choice here, as in [18], is the identity: $\bot = [x]\, \hat{x} \to x$. This allows in particular to catch matching failures, and then to trigger alternative or default cases. The result $b^{\{a/[\theta]\, p\}}$ is called **contractum** of r. For now, the rule can be applied in any context. The reduction ρ of a redex r in a term t is noted $\rho : t \xrightarrow{r}_{\beta_m} t'$. Annotations β_m, r and ρ can be omitted when the information is not needed.

Reduction inside *any context* is formalized as follows:

$$\frac{t_1 \xrightarrow{r}_{\beta_m} t_1'}{t_1 t_2 \xrightarrow{r}_{\beta_m} t_1' t_2} \qquad \frac{t_2 \xrightarrow{r}_{\beta_m} t_2'}{t_1 t_2 \xrightarrow{r}_{\beta_m} t_1 t_2'}$$

$$(\zeta)\ \frac{p \xrightarrow{r}_{\beta_m} p'}{[\theta]\, p \to b \xrightarrow{r}_{\beta_m} [\theta]\, p' \to b}$$

$$(\xi)\ \frac{b \xrightarrow{r}_{\beta_m} b'}{[\theta]\, p \to b \xrightarrow{r}_{\beta_m} [\theta]\, p \to b'}$$

Example 7.

Running Example. *Fragments of* Example 6 *are gathered here. Contracted redexes are underlined:*

$$\begin{array}{lll} \underline{([x_1 x_2]\, \hat{x_1}\hat{x_2} \to ([y]\, x_2\hat{y} \to b)\mathbf{c})\ (\mathbf{c}([z]\, \hat{z} \to z\mathbf{c}))} & & \text{(RE1)} \\ \quad \to_{\beta_m} \quad ([y]\, \underline{([z]\, \hat{z} \to z\mathbf{c})\hat{y}} \to b)\mathbf{c} & & \text{(RE2)} \\ \quad \to_{\beta_m} \quad \underline{([y]\, \hat{y}\mathbf{c} \to b)\, \mathbf{c}} & & \text{(RE3)} \\ \quad \to_{\beta_m} \quad \bot & & \text{(RE4)} \end{array}$$

Symbol ρ is for one-step reduction, and $\overrightarrow{\rho}$ for a sequence. A **normal form** is a term which can not be further reduced. A term t is **normalizable** if there exists $\overrightarrow{\rho} : t \to t'$ with t' a normal form. A term t is **terminating** or strongly normalizing if there is no infinite reduction from t.

Remark on expressivity. Note that λ-calculus enjoys a direct encoding into *PPC*: the λ-term $\lambda x.t$ can be rewritten as the *PPC*-term $[x]\, \hat{x} \to t$. To recover β-reduction $(\lambda x.t)u \to_{\beta} t^{\{x \mapsto u\}}$, remark that application $[x]\, \hat{x} \to t$ generates the trivial matching $\{u/[x]\, \hat{x}\}$ which results in the substitution $\{x \mapsto u\}$.

2.2 The Weak Pure Pattern Calculus

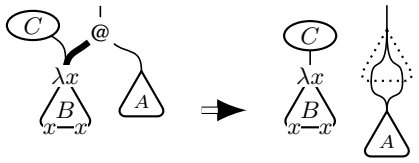
At run-time, evaluation of functional programs is mainly a matter of passing arguments to functions, and not evaluating functions bodies before they get instantiated. This leads to the study of **weak reduction**, where the reduction rule is never applied *under an abstraction*. This reduction relation is formalized by removing the (ξ)-rule of *PPC*. Remark that the (ζ)-rule concerning pattern reduction is kept, even if it's in some way *under an abstraction*. The reasons are that in the pattern, only matchables (which can't be substituted) are bound and not variables, and that reduction of the pattern may be *necessary* when a case is applied to an argument!

Restriction of contexts. As in λ-calculus, removing (ξ) breaks confluence. One solution preserving confluence [7] adopts a restricted rule which limits reduction under an abstraction to subterms independent of the abstracted variables. Replacing (ξ) by (ξ') defines ***WPPC***, the ***Weak Pure Pattern Calculus***.

$$(\xi')\ \frac{b \xrightarrow{r}_{\beta_m} b' \qquad \theta \# r}{[\theta]\, p \to b \xrightarrow{r}_{\beta_m} [\theta]\, p \to b'}$$

Now, given a preredex $r = ([\theta]\, p \to b)a$ in a term t, the definition of $\{a/[\theta]\, p\}$ is not enough for r to be a redex. It also requires r to be *closed in* t: variables free in r should not be bound outside. Remark that with the convention on names, condition $\theta \# r$ in (ξ') is equivalent to $\theta \cap fv(r) = \emptyset$.

Remark on the reduction under an abstraction. This definition of *WPPC* allows some reductions in the scope of an abstraction. This is required for full laziness. Indeed, consider some redex r in the dotted zone in the following picture (taken from the introduction). This redex is shared between B and the dotted zone, and hence can be reached by two paths: on the right its reduction can be required in order to reach a usual weak head normal form, whereas on the left it is seen as *under an abstraction*. And since these two versions of r are shared, they are reduced at the same time. Please notice that the mentioned weak head normal form is what a typical evaluator would try to reach and may not be a normal form of the weak calculus (but may need some further evaluation to be one). The point here is that full laziness may require some reduction under an abstraction even if this was not intended at the beginning.



Descendants and residuals. For a reduction $\rho : t \xrightarrow{r}_{\beta_m} t'$ and a subterm s_a of t, **descendants** of s_a after ρ, noted s_a/ρ, are the subterms s_d of t' that *come from* s_a. If $s_d \in s_a/\rho$, then call s_a an **ancestor** of s_d. Descendants after a one-step reduction are described in the enumeration below.
Write $r = ([\theta]\, p \to b)a \to_{\beta_m} r'$.

1. If s_a is disjoint from r, then s_a remains unchanged, and thus $s_a/\rho = \{s_a\}$.
2. If r is a strict subterm of s_a, that is $s_a = \mathcal{C}[r]$ with $\mathcal{C}[] \neq []$, then $s_a/\rho = \{\mathcal{C}[r']\}$.
3. If $\{a/[\theta]\, p\} = \sigma$ and s_a is a subterm of b but not a variable in θ, then $s_a/\rho = \{s_a^{\sigma}\}$.
4. If $\{a/[\theta]\, p\} = \sigma$ and s_a is a subterm of a which is in the codomain of σ, then let x be the variable and $\mathfrak{p}$ the position such that $\sigma_x|_{\mathfrak{p}} = s_a$. For each position $\mathfrak{q}$ such that $b|_{\mathfrak{q}} = x$, the subterm s_a has a descendant at position $\mathfrak{q}\mathfrak{p}$ in the descendant b^{σ} of b.
5. In any other case, s_a has no descendant: $s_a/\rho = \emptyset$.

An extension to general reductions is given by:

- For any set $\mathcal{S}$ of subterms, consider all descendants: $\mathcal{S}/\rho = \{s_d \mid \exists s_a \in \mathcal{S}, s_d \in s_a/\rho\}$.
- For any non-empty sequence of reduction $\overrightarrow{\rho}$, and for ρ_0 a one-step reduction, $\mathcal{S}/(\overrightarrow{\rho}\,\rho_0) = (\mathcal{S}/\overrightarrow{\rho})/\rho_0$.

The term **residual** denotes a redex which is the descendant of a redex. A redex r_c is **created** by ρ if it is not the descendant of a redex (which doesn't mean that r_c has no ancestor).

Example 8.
In Running Example 7, $([z]\ \hat{z} \to z\boldsymbol{c})\hat{y}$ *in term* (RE2) *is a descendant (in fact, the unique descendant) of* $x_2\hat{y}$ *from term* (RE1). *It is also a redex created by this reduction step (and not a residual) since the ancestor* $x_2\hat{y}$ *was not a redex. On the other hand, the application* $\boldsymbol{c}([z]\ \hat{z} \to z\boldsymbol{c})$ *of term* (RE1) *is destroyed by the matching and has no descendant.*

Developments. Let t be a term, and $\mathcal{R}$ be a set of redexes of t. A **development** of $\mathcal{R}$ is a sequence of reduction $\overrightarrow{\rho} = \rho_1...\rho_n$ that contracts only (residuals of) redexes of $\mathcal{R}$: for any $i \in 2...n$, the one-step reduction ρ_i contracts a redex in the set of residuals $\mathcal{R}/\rho_1...\rho_{i-1}$. A development $\overrightarrow{\rho}$ of $\mathcal{R}$ is **complete** when $\mathcal{R}$ has no descendant after $\overrightarrow{\rho}$, that is if $\mathcal{R}/\overrightarrow{\rho} = \emptyset$.

WPPC is confluent, and enjoys traditional properties of finite developments. These facts are corollaries of Confluence Theorem 2 and Finite Developments Theorem 4 on the labelled calculus defined in Section 3.

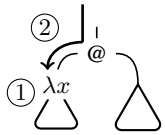
3. The Labelled Weak Pure Pattern Calculus

3.1 From Causality to Labels

One slogan of optimality theory [24] is *redexes with same origin should never be separated, and should therefore be reduced in one unique step*. The notion of origin of a redex r is to be understood here as the set of all past reduction events that were *necessary* for r to be a redex. Formal words for this are: the reduction steps that **contribute** to the creation of the redex r.

This subsection analyzes this contribution relation, and prepares the direct recording of contribution into terms via labels. The first step here is to characterize the cases of direct contribution, which correspond to the cases where a redex can be created.

Creation of redexes in λ-calculus has been studied in [25] and classified in three cases. For the purpose of this paper a coarser point of view is enough, and Lévy's classification is summed up in two cases:



In λ-calculus this redex is created when the abstraction comes in touch with the application. This can be due either to reduction of the left part of the application into an abstraction (1) (which covers the two first cases of Lévy) or to an external substitution replacing some variable occurrence with the abstraction (2) (which is exactly the third case of Lévy).

Example 9.
Creation case (1) *covers the two first cases of Lévy:*

$$(([\emptyset]\ \hat{c} \to ([\theta]\ p \to b))\hat{c})a \to ([\theta]\ p \to b)a$$
$$(([x]\ \hat{x} \to x)([\theta]\ p \to b))a \to ([\theta]\ p \to b)a$$

In the weak calculus studied here, a new case of creation arises: when the term has already the shape of a redex, but is not one due to an occurrence of an externally bound variable. The preredex becomes a redex when this occurrence is substituted. This is captured by a generalization of case (2) where a substitution acts anywhere inside the left or right part of the application, which is written (2^+) below.

Example 10.
Weak case: a preredex isn't a redex due to the only presence of some variables (ancestors of created redexes are underlined here).

$$([x]\ \hat{x} \to (\underline{([y]\ \hat{y} \to xy)a}))t \to ([y]\ \hat{y} \to ty)a$$

This also contains Lévy's third case:

$$([x]\ \hat{x} \to (\underline{xa}))([y]\ \hat{y} \to b) \to ([y]\ \hat{y} \to b)a$$

Example 11.
These two first cases (1) *and* (2^+) *already cover some characteristic features of PPC and pattern polymorphism.*
Handling of failure (1)*:*

$$(([\emptyset]\ \hat{c} \to ([\theta]\ p \to b))\hat{c}\hat{c})a \to \bot a = ([x]\ \hat{x} \to x)a$$

Instantiation of a pattern (2^+)*:*

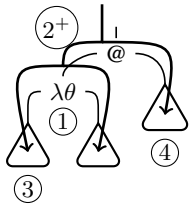
$$([x]\ \hat{x} \to (\underline{([y]\ x\hat{y} \to b)\hat{c}a}))\hat{c} \to ([y]\ \hat{c}\hat{y} \to b)\hat{c}a$$

In the following pictures, an abstraction $[\theta]\ p \to b$ is denoted by a binary node $\lambda\theta$ whose left son represents the pattern p and right son the body b, as below.

Pattern matching and dynamic patterns bring two symmetrical new cases of creation. Indeed, when the abstraction of a pattern p is applied to an argument a then a has to be matched against p. But, as seen in the previous section, $\{a/[\theta]\ p\}$ is not always defined. In particular some parts of a or p may be required to be in matchable form, which may in turn require further evaluation of a or p. In this case, the match $\{a/[\theta]\ p\}$ is defined only after some reductions in a or p, which means that a redex is created by reductions in the pattern or the argument.

Then the two new cases are: reduction in the pattern (3) and reduction in the argument (4). The only other case of non-matchable form is the presence of a variable occurrence that has to be substituted and falls in the case (2^+) of substitution, as shown in Example 11.

It is worth noticing that while case (3) is a specificity of dynamic patterns, case (4) already exists with static patterns, as illustrated in the introduction by Example 4.

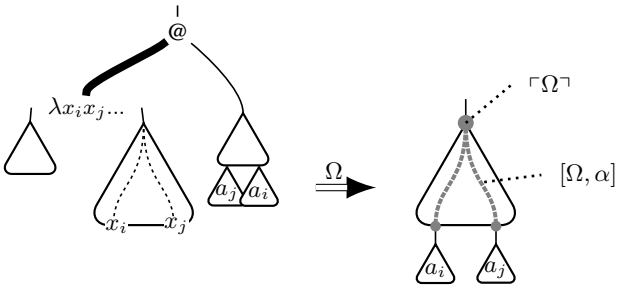


Example 12.
In Running Example 7*, the redex of term* (RE2) *is created by the first step along case* (2^+)*, whereas the redex of term* (RE3) *is created by the second step along case* (3).

A labelling system can be constructed by inverting the above description of redex creation: the question is not anymore *where does this redex come from?* but *what can be the future consequences of this reduction step?* The approach turns from *backward* to *forward*. If each reduced redex drops its fingerprint wherever it can possibly contribute to something (but nowhere else!), then the origin of a redex can be known by collecting the information left at this place by past reductions.

To achieve this, labels are added to the syntax of *WPPC*: every subterm bears a label recording any locally relevant information about past reductions, and reduction acts on these labels to keep contribution information up to date. For this, each redex gets a name deduced from neighbouring labels (in a way precised in Section 3.2), name which is used to track the future contributions of the redex. In such a framework the optimality commandment becomes more concrete by switching to: *reduce in one step all redexes with same name*.

The reduction of a redex r of name Ω transforms the labels of its contractum. Firstly r can contribute to something at its root through case (1): a label $\ulcorner\Omega\urcorner$ (denoting the *epicentre of* Ω) is added at the root of the contractum to witness this. Secondly if r is a successful match, then it generates a substitution. In this case r can contribute to something through case (2^+) anywhere along the propagation of the substitution. This defines a connected area of the contractum which is referred to in this paper as **substitution slice** (it is a slice in the sense of [32, Chap. 8]). The substitution slice is the union of the paths from the root of the contractum to each substituted variable occurrence. This other kind of contribution is witnessed in the labelling by an other construct: each atomic label α in the substitution slice is turned into another atomic label $[\Omega, \alpha]$ which can be understood as *a copy of* α *triggered by* Ω. This is summed up in the following picture.



Contributions from pattern or argument reduction are not visible here: indeed these contributions do not concern the contractum of the redex, but some other undetermined places in the context. The design of a mechanism taking into account these non local contributions is the main difficulty of the next section. To achieve this, the *forward* labelling aspects presented above are mixed with a *backward* analysis of the pattern and the argument: any relevant information found in the pattern or the argument of a redex is included in its name. The difficulty is then to discriminate between relevant and irrelevant parts of the pattern and the argument: they are not the same in case of success or failure of the pattern matching. Thus relevant prefixes of the pattern and the argument are mutually dependant and have to be dynamically determined by the pattern matching operation itself.

3.2 Formalizing Labels

This subsection defines the *Labelled Weak Pure Pattern Calculus* (*LWPPC*), which embeds a characterization of the aforementioned contribution relation.

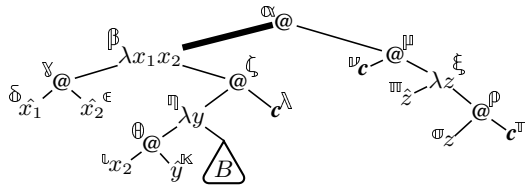
Syntax of *PPC* is extended with the labels introduced above. In the grammar, terms that must (resp. that must not) have a label in root position are called labelled (resp. clipped).

$$
\begin{array}{rcll}
A, B, P, R, T & ::= & \alpha : X & \text{Labelled terms} \\
X, Y, Z & ::= & T \mid N & \text{Terms} \\
N & ::= & x \mid \hat{x} \mid TT \mid [\theta]\, T \to T & \text{Clipped terms} \\
\alpha & ::= & \mathbb{\alpha} \mid \ulcorner\Omega\urcorner \mid [\Omega, \alpha] & \text{Atomic labels} \\
\Gamma, \Delta, \Omega & ::= & \alpha_1 \alpha_2 \ldots \alpha_n & \text{Labels}
\end{array}
$$

where n is a strictly positive integer, $x, y, z \in \mathcal{X}$ the set of names and $\mathbb{\alpha}, \mathbb{\beta} \ldots \in \mathbb{A}$ the set of initial labels (blackboard bold Greek letters are used to distinguish initial labels when needed). An **initial term** is a term whose labels are all initial and different. Remark that this calculus is not labelled in the sense of [32]: labels are not used as a decoration for function symbols but added to the syntax as in [4].

For any possibly non-atomic label $\Gamma = \alpha_1 \ldots \alpha_n$, write $\Gamma \cdot X$ as a shorthand for $\alpha_1 : \ldots : \alpha_n : X$. Letters P, B and A are still used for terms playing the role of pattern, function body and argument, while the Greek letter Ω indicates the name of a redex. Natural notion of position is still used, but now taking into account atomic labels. Notions of free variables, free matchables, α-conversion and substitution are inherited from *PPC*.

Example 13.
Term (RE1) *of* Running Example 7 *with an initial labelling (referred to as* (L1) *in future examples).*



The **direct contribution** $\prec$ is a well-founded relation over labels which turns out to be useful in following sections. The notion of direct contribution appears in [28] for term rewriting systems. It is adapted for *LWPPC* as follows: $\Omega \prec \Gamma$ if and only if $\Gamma = \Gamma_1 \ulcorner\Omega\urcorner \Gamma_2$ or $\Gamma = \Gamma_1 [\Omega, \alpha] \Gamma_2$ (both Γ_1 and Γ_2 may be empty here). Remark in this definition that Ω and α do not play symmetrical roles in $[\Omega, \alpha]$.

The following grammar extends notions of data structure and matchable form to labelled terms.

$$
\begin{array}{rcll}
D_l & ::= & \alpha : D & \text{Labelled data structures} \\
D & ::= & \hat{x} \mid D_l T \mid D_l & \text{Data structures} \\
M & ::= & D \mid [\theta]\, P \to B \mid \alpha : M & \text{Matchable forms}
\end{array}
$$

To any matchable form M is associated a label $|M|$ called **matchability witness** which is meant to record past events that contributed to put M in matchable form.

$$
\begin{array}{rcl}
|[\theta]\, P \to B| & := & \varepsilon \\
|\hat{x}| & := & \varepsilon \\
|T_1 T_2| & := & |T_1| \\
|\alpha : Z| & := & \alpha |Z|
\end{array}
$$

The labelled compound matching returns the pair of a label Δ and a match μ. The label Δ is meant to collect on-the-fly any contribution information relative to the evaluation of the pattern and/or the argument.

For correct treatment of matchability witnesses, the labelled compound matching has two policies: **simple** compound matching $\{\!\{Y/[\theta]\, X\}\!\}_{\mathrm{s}}$ records all the labels inside Y and X that are visited during the matching operation, whereas **witnessing** compound

matching $\{\!\{Y/[\theta]\,X\}\!\}_w$ also records the matchability witness of Y even if Y isn't completely visited. In $\{\!\{Y/[\theta]\,X\}\!\}_p$ the subscript p is either s or w. Notation $\alpha : (\Delta, \mu)$ stands for $(\alpha\Delta, \mu)$.

$$
\begin{array}{rcll}
\{\!\{Y/[\theta]\,\alpha : Z\}\!\}_p & := & \alpha : \{\!\{Y/[\theta]\,Z\}\!\}_p & \\
\{\!\{Y/[\theta]\,\hat{x}\}\!\}_s & := & (\varepsilon, \{x \mapsto Y\}) & x \in \theta \\
\{\!\{Y/[\theta]\,\hat{x}\}\!\}_w & := & (|Y|, \{x \mapsto Y\}) & x \in \theta \\
\{\!\{Y/[\theta]\,([\tau]\,P \to B)\}\!\}_s & := & (\varepsilon, \bot) & \\
\{\!\{Y/[\theta]\,([\tau]\,P \to B)\}\!\}_w & := & (|Y|, \bot) & \\
\{\!\{\alpha : Z/[\theta]\,X\}\!\}_p & := & \alpha : \{\!\{Z/[\theta]\,X\}\!\}_p & \\
 & & X \text{ matchable form, otherwise} & \\
\{\!\{\hat{x}/[\theta]\,\hat{x}\}\!\}_p & := & (\varepsilon, \{\}) & x \notin \theta \\
\{\!\{A_1A_2/[\theta]\,P_1P_2\}\!\}_p & := & \{\!\{A_1/[\theta]\,P_1\}\!\}_w \Subset \{\!\{A_2/[\theta]\,P_2\}\!\}_s & \\
 & & A_1A_2,\, P_1P_2 \text{ matchable forms} & \\
\{\!\{Y/[\theta]\,X\}\!\}_p & := & (|Y||X|, \bot) & \\
 & & Y, X \text{ matchable forms, otherwise} & \\
\{\!\{Y/[\theta]\,X\}\!\}_p & := & \texttt{undefined} & \text{otherwise}
\end{array}
$$

where $\Subset$ is defined by the following table (with $\uplus$ the usual disjoint union of substitutions).

$\Subset$	(Δ_2, σ_2)	$(\Delta_2, \bot)$	$\texttt{undefined}$
(Δ_1, σ_1)	$(\Delta_1\Delta_2, \sigma_1 \uplus \sigma_2)$	$(\Delta_1\Delta_2, \bot)$	$\texttt{undefined}$
$(\Delta_1, \bot)$	$(\Delta_1, \bot)$	$(\Delta_1, \bot)$	$(\Delta_1, \bot)$
$\texttt{undefined}$	$\texttt{undefined}$	$\texttt{undefined}$	$\texttt{undefined}$

This operation is asymmetrical, due to the following difficulty: if an argument is incompatible with the pattern for two distinct reasons, which one should be blamed for it? In *PPC* any choice is fine, but with labelled terms distinct culprits yield distinct labels, and confluence is broken. This fact can be understood from a higher point of view by noticing that labelling of non-orthogonal calculi may lead to non-confluence [32, Chap. 8]. And as pointed out in [23], allowing matching failures as it is done in *PPC* breaks orthogonality (although the system still seems weakly orthogonal).
Remembering both failures is an easy but unsatisfying way to preserve confluence: it forces the matching operation to explore completely the pattern even if a failure has been detected, which doesn't seem to be a great solution when looking for an efficient evaluation model. The solution preferred here for solving this tricky problem is to set an order for pattern matching operations (the left-to-right definition of $\Subset$ is just an example). Unfortunately, this causes a restriction on allowed reduction strategies. Any deterministic order being acceptable for each pattern matching, interesting degrees of freedom are preserved, but the way this could be used for optimization is unclear.

Remark that $\{\!\{Y/[\theta]\,X\}\!\}_w$ (with witnessing policy) is ill-defined if Y isn't a matchable form (in this case $|Y|$ has no sense). But if starting with simple policy, this case never happens.

Example 14.
For the redex in labelled term (L1) *(*Example 13*), the simple compound matching succeeds and returns the substitution* $\{x_1 \mapsto \mathbb{v} : \boldsymbol{c},\ x_2 \mapsto \xi : ([z]\,\mathbb{m} : \hat{z} \to \mathbb{p} : (\mathbb{w} : z)(\mathbb{r} : \boldsymbol{c}))\}$ *and the label* $\Delta = \mathbb{y}\mathbb{u}\ \mathbb{\delta v}\ \epsilon$.

Given a label Ω and a list θ of names, the **relabelling** $\Omega(\!|\theta|\!)X$ of X is defined as X if $\theta\#X$ or X is a variable. Otherwise:

$$
\begin{array}{rcll}
\Omega(\!|\theta|\!)(T_1T_2) & := & (\Omega(\!|\theta|\!)T_1)(\Omega(\!|\theta|\!)T_2) & \\
\Omega(\!|\theta|\!)([\tau]\,P \to B) & := & [\tau]\,(\Omega(\!|\theta|\!)P) \to (\Omega(\!|\theta|\!)B) & \theta \cap \tau = \emptyset \\
\Omega(\!|\theta|\!)(\alpha : Z) & := & [\Omega, \alpha] : (\Omega(\!|\theta|\!)Z) &
\end{array}
$$

A **labelled preredex** is a labelled term of the form

$$R = \alpha : ((\Gamma \cdot [\theta]\,P \to B)A)$$

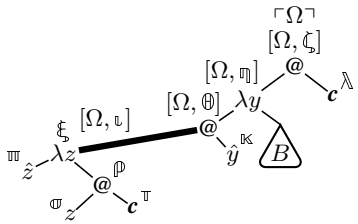
Define $\{A/[\theta]\,P\}$ to be the check of $\{\!\{A/[\theta]\,P\}\!\}_s$ against θ (default policy is the simple one). R is a **labelled redex** when it satisfies the restriction for weak reduction and $\{A/[\theta]\,P\} = (\Delta, \mu)$. Then the redex R get the name $\Omega = \alpha\Gamma\Delta$, and the rule is:

$$R \quad \xrightarrow{R} \quad \ulcorner\Omega\urcorner : (\Omega(\!|\theta|\!)B)^{\mu}$$

where as previously $X^{\bot} = \bot$ for any term X, with $\bot$-as-a-term a fixed closed normal form, for instance the identity with some distinguished initial labels:

$$\bot \quad = \quad [x]\,(\mathbb{m}_{\bot} : \hat{x}) \to (\mathbb{\beta}_{\bot} : x)$$

Example 15.
Reduction of (L1) *of* Example 13 *results in the following term, where* $\Omega = \mathbb{a}\ \mathbb{\beta}\ \mathbb{y}\mathbb{u}\mathbb{\delta v}\epsilon$



For any set $\mathcal{N}$ of labels, let $\to_{\mathcal{N}}$ be the reduction relation restricted to redexes with names in $\mathcal{N}$.

$$\frac{Z \ \xrightarrow{R}_{\mathcal{N}} \ Z'}{\alpha : Z \ \xrightarrow{R}_{\mathcal{N}} \ \alpha : Z'}$$

$$\frac{T_1 \ \xrightarrow{R}_{\mathcal{N}} \ T_1'}{T_1T_2 \ \xrightarrow{R}_{\mathcal{N}} \ T_1'T_2} \qquad \frac{T_2 \ \xrightarrow{R}_{\mathcal{N}} \ T_2'}{T_1T_2 \ \xrightarrow{R}_{\mathcal{N}} \ T_1T_2'}$$

$$\frac{P \ \xrightarrow{R}_{\mathcal{N}} \ P'}{[\theta]\,P \to B \quad \xrightarrow{R}_{\mathcal{N}} \quad [\theta]\,P' \to B}$$

$$\frac{B \ \xrightarrow{R}_{\mathcal{N}} \ B' \qquad \theta\#R}{[\theta]\,P \to B \quad \xrightarrow{R}_{\mathcal{N}} \quad [\theta]\,P \to B'}$$

Notion of descendant is straightforwardly extended, and developments follow. Remark that there are new subterms without ancestors in *LWPPC*: if $R = \alpha : (\Gamma \cdot ([\theta]\,P \to B))A \to \ulcorner\Omega\urcorner : (\Omega(\!|\theta|\!)B)^{\sigma} = R'$, then $(\Omega(\!|\theta|\!)B)^{\sigma}$ is a descendant of B, but R' itself has no ancestor, due to the created label $\ulcorner\Omega\urcorner$.

Cases of redex creation are the same as those of *WPPC*.

3.3 Properties of LWPPC

The following lemma states the main property of residuals and justifies the *development of a name* introduced in Section 4.

Lemma 1 (Redex Stability). *Let X be a term with a redex R_a of name Ω_a. If R_d is a descendant of R_a after a reduction ρ, then R_d is still a redex of name Ω_a (and hence R_d is a residual of R_a).*

Proof. Based on properties of stability of matchability witnesses and labelled compound matching under reduction. □

Remark that Lemma 1 does not hold if reduction is allowed in any context. Indeed if the context rule (ξ) is kept, the descendant of a redex is still a redex, but its name can change. This happens when a redex is rooted in the substitution slice of the reduced redex, which is precisely what is forbidden by the restricted context rule (ξ'). Hence this crucial Redex Stability Lemma 1 holds only for the weak calculus considered here, as well as all subsequent main results.

Theorem 2 (**Confluence).** LWPPC *is confluent.*

Proof. Confluence of $\rightarrow_{\mathcal{N}}$ for any set of labels $\mathcal{N}$ is proved by using the Tait and Matin-Löf's technique, with the Redex Stability Lemma 1 as cornerstone. □

Theorem 3 (**Termination).** *For any finite set of labels $\mathcal{N}$, $\rightarrow_{\mathcal{N}}$ is strongly normalizing.*

Proof. The proof is immediate with the following lemma: if a term T and a substitution σ are strongly normalizing, then $(\Omega \, \langle\!\langle dom(\sigma) \rangle\!\rangle \, T)^{\sigma}$ is strongly normalizing for any Ω. The latter is proved using a lexicographic product of the direct contribution relation (reversed) and the subterm relation. □

Theorem 4 (**Finite Developments).** *For any term X, set $\mathcal{R}$ of redexes and set $\mathcal{S}$ of subterms:*

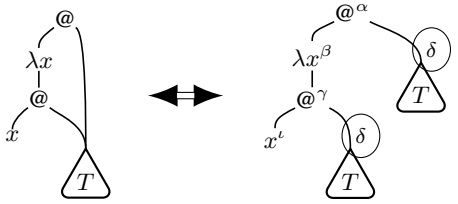
- *Any development of $\mathcal{R}$ is finite.*
- *All complete developments of $\mathcal{R}$ yield the same result.*
- *The set of descendants of $\mathcal{S}$ is the same after any complete development of $\mathcal{R}$.*

Proof. As usual, the proofs of properties on developments use some *marking* to track residuals. However, instead of defining a new marked calculus, the internal labelling system of *LWPPC* can be directly used by introducing a fresh initial label for each subterm to be traced. Through this trick, the results on developments are essentially consequences of Termination Theorem 3 and Confluence Theorem 2. □

LWPPC gives instructions for optimal sharing in *WPPC* by a description of what should be reduced in one single step. But as stated for now these instructions are not constructive. Next section shows how labels can be used to derive an effective graph implementation of sharing.

4. The Sharing Property

The previous section introduces *LWPPC*, a variant of *WPPC* where each term and subterm bears a label (an atomic label or a sequence). The labelling describes a graph implementation with the following idea: each atomic label represents a *memory location*. Subterms with same label are thus meant to be *physically equal*. This idea is specified for first order terms in [9, 28], and for weak λ-calculus in [4].



Of course not all labelled terms can be translated to graphs in a consistent way. The **sharing property** $\mathbb{S}$ is a formal condition allowing it: a term T is said to enjoy the sharing property (noted $\mathbb{S}(T)$) if for any of its subterms $\alpha_1 : Z_1$ and $\alpha_2 : Z_2$,

$$\alpha_1 = \alpha_2 \quad \text{implies} \quad Z_1 = Z_2$$

Main issue is now to check that correspondence with graphs is not only static but also dynamic, by defining a notion of reduction over labelled terms which preserves the sharing property and corresponds to the natural graph reduction.

The **development of a name** $\Rrightarrow$ is defined as follows: given terms T, T', write $T \Rrightarrow T'$ if there exists a label l such that T' is the result of the complete development of the redexes of T with name l (all developments are equivalent by Finite Developments Theorem 4). To prove that this reduction preserves sharing, the following key property of direct contribution is needed:

Lemma 5 (**Direct Contribution).** *For any reduction $T \xrightarrow{R} T'$ with redex R of name Ω, if R_c is a redex of T' with name Ω_c created by the reduction, then $\Omega \prec \Omega_c$.*

Proof. By case on the redex creation. □

Theorem 6 (**Preservation of Sharing).** *Let $T \Rrightarrow^* T'$ with T an initial term. Then T' enjoys the sharing property.*

Proof. Similar to the proof presented in [4]. A key point is an invariant stating that names of redexes are maximal for the direct contribution relation. The proof makes also an important use of the Direct Contribution Lemma 5. □

Outcome of this part is a graph implementation of weak pure pattern calculus featuring optimal sharing, which corresponds to fully lazy sharing for *PPC*: when a function body is instantiated, the only modified labels (which means the only copied nodes!) correspond to the substitution slice.

5. The Result of Optimality

Now that an implementation model is defined, this section characterizes a family of strategies over graphs (resp. labelled terms) which always normalize in a minimal number of reduction steps (resp. developments of names). A straightforward corollary will be that these strategies are correct: whenever a normal form exists (unique, by Confluence Theorem 2), they reach it (with a minimal number of steps). The discussion here is informal, and puts the focus on the following message: all the tough work toward an optimality result has already been done in previous sections.

A redex R in a term T is said to be **needed** when any reduction $\overrightarrow{\rho} : T \rightarrow T'$ to a normal form T' contracts R or at least one of its residuals. A needed strategy reduces only needed redexes.

An axiomatic framework making needed strategies optimal is given in [12]. Ingredients are: a notion of residual, a family relation, a contribution relation over families (families are the equivalence classes of the family relation), and a set of terms considered as *results*, each satisfying its own group of axioms. This kind of result is mostly inapplicable if one doesn't know how to find such abstract notions in the concrete system, but the labels of *LWPPC* provide a solution:

- The extant notion of residual is used (Section 2).
- Two redexes are defined to be in the same family if and only if they have the same name (hence each family is assimilated to a name).
- Define the abstract contribution relation as the extant direct contribution relation on labels (Section 3.2).

To form a *Deterministic Family Structure* (that is the name of the abstract concept), these definitions have to satisfy finite development properties (Finite Development Theorem 4), finite family developments (deduced from Termination Theorem 3), and properties relating contribution relation to creation of redexes (deduced from Redex Stability Lemma 1 and a converse property).

Finally, the set of weak normal forms is an easy example of stable set of results, and hence results of [12] apply.

Corollary 7 (**Correctness & Optimality).** *Let T be an initial normalizable term. Then any needed reduction of $\Rrightarrow$ reaches the (unique) normal form with a minimal number of steps.*

6. Related Works

The approach used in this paper to derive a graph implementation owes a lot to Blanc, Lévy and Maranget, who described a labelled weak λ-calculus enjoying the sharing property [4]. Their work is generalized here in several ways. First the method used to derive a labelling system is guided by the notion of contribution. This approach enables a slight simplification of their labels (only two syntactic constructs versus three, and also less indirections). Secondly the approach is extended to get results on strategies and not only on the representation of programs. Last but not least, the study is done on a pattern calculus which is a strict generalization of λ-calculus.

Labelled terms representing graphs and results of correctness and optimality are studied by Maranget in [28] for orthogonal first-order term rewriting systems, and even applied to pattern matching *à la ML* through compilation into supercombinators. However, this compilation scheme makes use of the static nature of patterns in Maranget's framework, and hence can not be applied to dynamic patterns. Moreover, such compilation treats pattern matching *a priori* and prevents the direct study of its history in a higher-order setting. Last, please notice that the labelling systems in [28] implement only the sharing of lazy evaluation, and that something new is needed for fully lazy evaluation, as suggested by [4]. Indeed full laziness is closely related to the particular weak reduction used here, which asks for a particular treatment of the substitution slice of each contractum. This can be done either by using as many rules as there are possible substitution slices (that means infinitely many), or by redefining a labelled substitution (which is the solution used here).

A lot of results on confluence, developments, descendants and origin tracking also exist in general (higher-order) rewriting frameworks [28, 32], and an encoding of *PPC* into *Combinatory Reduction Systems (CRS)* in particular is suggested by Klop, van Oostrom and de Vrijer in [23]. Unfortunately this encoding has one major drawback from the implementational point of view. Indeed it is based on a rule scheme that generates one rule for each term acceptable as a pattern (that is for successful matchings) and has to be extended with more complex schemes for matching failures. This enumeration of all possible matchings leads to a *CRS* with infinitely many rules which can be useful for understanding key notions of origins [32] and providing immediate proofs of some results (such as confluence [33] or finite family developments [6]), but seems difficult to be used as such for an implementation. Moreover, remark that infinitely many rules yield an infinite alphabet of rule names for labelling, whereas in *LWPPC* all is built with one rule and the finite set of labels that decorates the original term.

More generally, a term (or a higher-order) rewriting system asks for an infinite enumeration of rules to model matching against all possible patterns. Each rule has a fixed shape which determines the relevant parts of the argument and the way labels are handled. On the other hand, pattern matching calculi such as *PPC* use for all patterns a unique matching algorithm which identifies dynamically the parts of the pattern and of the argument that are relevant for matching. This leads to a new view on labelling which is addressed in this paper.

The optimality result could also have been proved in a more standard way [29, 35], namely by stating a one-step diamond property for the development of names in *LWPPC* (which can be proved using the same central result: Finite Developments Theorem 4). The abstract approach by Glauert and Khasidashvili [12] is preferred here for its modularity. It has already been successfully applied on wide classes of higher-order rewriting systems, and could work on an encoding of *PPC* by using general results of [6, 32]. An alternative method is proposed here, which is a direct reuse of the general results on *LWPPC* (and hence is more self-contained).

Also notice that *PPC* has already an implementation, known as the programming language Bondi developed by Jay [5] [17, Part 3]. The graph implementation presented here could help improving the efficiency of the Bondi evaluator by introducing more sharing.

7. Conclusion and Prospects

Enriched pattern matching paradigms allowing dynamic patterns, such as *PPC*, improve expressive power and usability of functional programming languages by offering path and pattern polymorphisms to the programmer. Unfortunately they also invalidate usual optimizations performed by compilers on functions defined by cases, which is a severe drawback when it comes to implementation.

This paper lays the foundations of a sharing theory for these frameworks, with the goal of overcoming this difficulty. Originality of this work is in the way a careful analysis of the contribution relation between redexes can be used to derive a graph implementation as well as optimal reduction strategies. This work is also the first application of such an analysis to a pattern matching framework based on a concise definition of matching instead of an enumeration of all possible matchings. Difficulties specific to this feature such as non-local contributions and the handling of failures are tackled.

The first result is a graph implementation featuring fully lazy sharing (Preservation of Sharing Theorem 6), which combines a fairly good level of sharing with the possibility of an efficient implementation. This result is associated with a description of reduction strategies that are correct and efficient (Correctness & Optimality Corollary 7).

However, this paper is just a first step toward a more ambitious program: as mentioned in the introduction the aim of this work is to provide sharing mechanisms that share pattern matching steps. This kind of sharing is limited in the current graph implementation, due to the implicit treatment of pattern matching: in *PPC* each matching is performed globally as an atomic operation, and can be shared only as a whole. This hides the fact that any matching is composed of several elementary matching steps which could be shared individually.

In order to get this finer control, the technology presented in this paper has to be extended to the *Explicit Pure Pattern Calculus* [2]: a variant of *PPC* with explicit pattern matching, where all matching steps are visible. In this extended framework every single piece of a pattern could be shared independently of the other parts, and if two patterns in the same function share some structure, then corresponding pattern matching steps could also be shared! Furthermore, to ensure that sharing is not lost along sequences of pattern matching cases, an explicit *alternative case* operator has to be added to the calculus (as it is done in the *Extension Calculus* [17]). A third point that has to be addressed is the analysis of the consequences of the requirement of a deterministic strategy for pattern matching operations. Few strategies are available for now, but explicit matching will enrich them.

This allows to hope for a functional programming language which would feature dynamic patterns and still have an efficient implementation: the sharing model could be used directly to manage pointers and copies in an efficient graph implementation such as [30], or to guide the design of an advanced implementation by interaction nets in the style of [10, 26].

The *explicit matching* framework will enrich also the optimality result. Firstly, an optimal strategy taking into account every single pattern matching operation is a way to manage all the low-level tests induced by functions defined by multiple cases. Secondly, whereas *PPC* only allows an abstract definition of needed strategies, some of them can be *effectively* described as variants of *leftmost-outermost* in the explicit framework.

Acknowledgments

Special thanks to my advisor Delia Kesner. Many thanks also to Luc Maranget, Zurab Khasidashvili, Maribel Fernández, Barry Jay and anonymous referees for helpful comments and suggestions on this work.

References

[1] A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*. Cambridge University Press, 1998.

[2] T. Balabonski. Calculs avec motifs dynamiques, Rapport technique PPS, Université Paris Diderot, 2008. Available at `http://hal.archives-ouvertes.fr/hal-00476940`.

[3] T. Balabonski. Optimality for Dynamic Patterns, Rapport technique PPS, Université Paris Diderot, 2010.

[4] T. Blanc, J.-J. Lévy, and L. Maranget. Sharing in the Weak Lambda-Calculus Revisited. In *Reflections on Type Theory, Lambda Calculus and the Mind* Essays Dedicated to Henk Barendregt on the Occasion of his 60th Birthday, December 2007.

[5] Bondi. Bondi programming language. `http://bondi.it.uts.edu.au/`.

[6] S. Bruggink. *Equivalence of Reductions in Higher-Order Rewriting*. Ph.D. thesis, 2008.

[7] N. Çağman and J. R. Hindley. Combinatory Weak Reduction in Lambda Calculus. *Theor. Comput. Sci.*, 198(1-2):239–247, 1998.

[8] H. Cirstea. *Rewriting Calculus: Foundations and Applications*. Ph.D. thesis, 2000.

[9] D. Dougherty, P. Lescanne, L. Liquori, and F. Lang. Addressed Term Rewriting Systems: Syntax, Semantics, and Pragmatics: Extended Abstract. *ENTCS*, 127(5):57–82, 2005.

[10] M. Fernández, I. Mackie, S. Sato, and M. Walker. Recursive Functions with Pattern Matching in Interaction Nets. *ENTCS*, 253(4):55–71, 2009.

[11] F. L. Fessant and L. Maranget. Optimizing Pattern Matching. In *ICFP*, pages 26–37, 2001.

[12] J. Glauert and Z. Khasidashvili. Relative Normalization in Deterministic Residual Structures. In *CAAP*, pages 180–195, 1996.

[13] R. Hinze and J. Jeuring. Generic Haskell: Practice and Theory. In *Generic Programming*, volume 2793 of *LNCS*, pages 1–56, 2003.

[14] C. Holst and D. Gomard. Partial Evaluation is Fuller Laziness. *SIGPLAN Not.*, 26(9):223–233, 1991.

[15] G. Huet and J.-J. Lévy. Computations in Orthogonal Rewriting Systems. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic, Essays in Honor of Alan Robinson*, pages 394–443. MIT Press, 1991.

[16] B. Jay. The Pattern Calculus. In *TOPLAS*, volume 26(6), pages 911–937, 2004.

[17] B. Jay. *Pattern Calculus: Computing with Functions and Data Structures*. Springer, 2009.

[18] B. Jay and D. Kesner. First-class patterns. *J. Funct. Programming*, 19 (2):191–225, 2009.

[19] B. Jay and D. Kesner. Pure Pattern Calculus. In *ESOP, LNCS 3942*, pages 100–114, 2006.

[20] B. Jay and D. Kesner. Patterns as First-Class Citizens. Technical report, 2008. Available as `http://hal.archives-ouvertes.fr/hal-00229331/fr/`.

[21] S. P. Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, Inc., 1987.

[22] J. W. Klop. *Combinatory Reduction Systems*. Ph.D. thesis, 1980.

[23] J. W. Klop, V. van Oostrom, and R. de Vrijer. Lambda Calculus with Patterns. *TCS*, 398:16–31, 2008.

[24] J.-J. Lévy. Optimal Reductions in the Lambda-Calculus. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalisms*, pages 159–191, 1980.

[25] J.-J. Lévy. *Réductions correctes et optimales dans le lambda-calcul*. Ph.D. thesis, 1978.

[26] I. Mackie. Efficient Lambda-Evaluation with Interaction Nets. In *RTA*, pages 155–169, 2004.

[27] L. Maranget. Compiling Pattern Matching to Good Decision Trees. In *ML*, pages 35–46, 2008.

[28] L. Maranget. *La stratégie paresseuse*. Ph.D. thesis, 1992.

[29] L. Maranget. Optimal Derivations in Weak Lambda-calculi and in Orthogonal Terms Rewriting Systems. In *POPL*, pages 255–269, 1991.

[30] O. Shivers and M. Wand. Bottom-up β-reduction: Uplinks and λ-DAGs. Technical Report RS-04-38, BRICS, December 2004.

[31] F.-R. Sinot. Complete Laziness: a Natural Semantics. *ENTCS*, 204: 129–145, 2008.

[32] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

[33] V. van Oostrom and F. van Raamsdonk. Weak Orthogonality Implies Confluence: the Higher-Order Case. In *LFCS'94*, pages 379–392, 1994.

[34] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. Ph.D. thesis, 1971.

[35] N. Yoshida. Optimal Reduction in Weak-λ-calculus with Shared Environments. *Journal of Computer Software*, 11(5):2–20, September 1994.