South American Journal of Logic
Vol. 1, n. 1, pp. 63–110, 2015
ISSN: 2446-6719

Combinatory logics for lambda calculi with patterns

Ariel Arbiser and Gabriela Steren

Abstract

Pattern matching is a basic building block on which functional programming depends, where the computation mechanism is based on finding a correspondence between the argument of a function and an expression called "pattern". It has also found its way into other programming paradigms and has proved convenient for querying data in different formats, such as semi-structured data. In recognition of this, a recent effort is observed in which pattern matching is studied in its purest form, namely by means of *pattern calculi*. These are lambda calculi with sophisticated forms of pattern matching. We propose to contribute to this effort by developing a combinatory logic for one such pattern calculus, namely λP. We seek to mimic the computational process of λP where arguments can be matched against *arbitrary* terms, *without* the use of variables. Two challenges must be met. On the one hand, dealing with bound variables in patterns. Indeed, an abstraction is a valid pattern in λP. Here the standard combinatory logic will provide guidance. The second is computing the counterpart, in the combinatory setting, of the substitution that is obtained in a successful match. This requires devising rules that pull applications apart, so to speak. We propose a combinatory logic that serves this purpose and study its salient properties and extensions including a typed presentation, and the introduction of constructors for modeling data structures and different possibilities for the matching mechanism, characterizing a family of confluent variants.

Keywords: rewriting, lambda calculus, combinatory logic, pattern, matching.

1 Introduction

Modern functional programming practice involves an extensive use of patterns. A pattern is basically a syntactic specification of a family of terms, which facilitates function definition by cases, currently a common and useful practice in declarative programming.

The λ-calculus, introduced in the 1930s and studied since then to this date, is considered a formal basis for functional programming. Having a remarkably simple and

concise syntax, it presents a wide variety of problems which are common in many modern programming languages and paradigms, and thus has a transcendental importance from the theoretical point of view. This is why research is currently being done for it and its many variations.

Pattern matching is a basic building block on which all modern functional programmers depend. A simple example of a program that relies on patterns is the `length` program for computing the length of a list (the code is written in Haskell).

```
length []     = 0
length (x:xs) = 1 + length xs
```

Another example is the following, which exhibits the use of nested patterns, `Salary` being a data constructor having an employee id and his/her salary as arguments:

```
update []                         f = []
update ((Salary id amount):xs) f = (Salary id (f amount)): update xs f
```

In recognition of their usefulness and in an effort to study patterns and their properties in their most essential form, a number of so called **pattern calculi** have recently emerged [9, 18, 32, 23, 2, 20, 21]. As a representative example, we briefly describe the pioneering λP [30] recently revisited in [25] (cf. Section 2 for detailed definitions). In λP the standard functional abstraction $\lambda x.M$ is replaced by the more general $\lambda P.M$:

$$M, N, P ::= x \mid MN \mid \lambda P.M$$

The pattern P may be any term at all. In particular, it may be a variable, thus subsuming the standard functional abstraction of the λ-calculus. Examples of λP-terms are $\lambda(\lambda x.y).y$, $\lambda z.\lambda(\lambda x.x).\lambda y.y$, $(\lambda(\lambda x.y).y)(\lambda w.z)$, and of course all the terms of the λ-calculus. The pattern specifies which form the argument must have in order to match. A function can only be applied to an argument which is an instance of its pattern. Application is then performed by substituting the terms bound to the free variables in the pattern into the function body:

$$(\lambda P.M)P^\sigma \to_{\beta_P} M^\sigma$$

Here σ denotes a substitution from variables to terms, and P^σ the result of applying it to P. Note that in the case that P is a variable, we obtain $\to_\beta$. An example of a λP-reduction step is:

$$(\lambda xy.x)((\lambda x.x)(zw)) \to_{\beta_P} \lambda x.x \tag{1}$$

Another example is the reduction step from the term $(\lambda(\lambda x.y).y)(\lambda z.w)$. The pattern $\lambda x.y$ can be matched by any constant function, and the result of the application of $\lambda(\lambda x.y).y$ to an argument of the form $\lambda z.M$ (with $z \notin \mathsf{FV}(M)$) will be M (in this case, $M = w$). However, the application $(\lambda(\lambda x.y).y)z$ does not reduce, since the argument z

does not match the pattern $\lambda x.y$. Note that $(\lambda(\lambda x.y).y)(\lambda z.z)$ does not reduce either, since $\lambda z.z$ is not an instance of $\lambda x.y$ (variable capture is not allowed). This situation, where reduction is *permanently* blocked due to the argument of an application not matching the pattern, is known as **matching failure**. There are cases where an argument does not yet match a pattern, but can reduce to a term which does. For example, $(\lambda(\lambda x.y).y)\,((\lambda x.x)\,(\lambda z.w))$. In this case, we say that the matching is still **undecided**.

A key issue is establishing the conditions under which confluence holds for λP, since the unrestricted calculus is not confluent. For example, the following reduction starts from the same λP-expression as (1) but ends in a different normal form:

$$(\lambda xy.x)((\lambda x.x)(zw)) \to_{\beta_P} (\lambda xy.x)(zw) \to_{\beta_P} z$$

In the same way that *CL* shows how one may compute without variables in the λ-calculus, we seek to address a similar property for *pattern calculi*. This would entail that variables and substitution in pattern calculi may be compiled away while preserving the reduction behavior. For this purpose we fix λP as study companion and delve into the task of formulating a corresponding combinatory logic.

In order to motivate our combinatory system CL_P, recall the standard translation from the λ-calculus to *CL*:

$$\begin{array}{rcl} x_{CL} & \triangleq & x \\ (MN)_{CL} & \triangleq & M_{CL}N_{CL} \\ (\lambda x.M)_{CL} & \triangleq & [x].M_{CL} \end{array}$$

where "$\triangleq$" denotes definitional equality and $[x].M$ is defined recursively as follows:

$$\begin{array}{lcll} [x].x & \triangleq & SKK & \\ [x].M & \triangleq & KM, & \text{if } M = K, S, I \text{ or } M = y \neq x \\ [x].(MM') & \triangleq & S([x].M)([x].M') & \end{array}$$

Note that abstractions translate to terms of the form KM or SM_1M_2. Therefore, the application of an abstraction to an argument will translate to either KMN or SM_1M_2N with N being the translation of the argument. In CL_P, applications of an abstraction to an argument in λP will translate to terms of the form K_PMN or $S_PM_1M_2N$, N being the translation of the argument. The expressions K_P and S_P are combinators of CL_P. The full grammar of CL_P-expressions is:

$$M, N ::= x \mid K_M \mid S_M \mid \Pi^1_{MN} \mid \Pi^2_{MN} \mid MN$$

Subscripts of combinators in CL_P are an integral part of them. Combinators K_P and S_P will behave similarly to K and S but with one important difference: while K and S always form redexes when applied to the right number of arguments (2 and 3 respectively), K_P (resp. S_P) will check its second (resp. third) argument against the

pattern P, and will only form a redex if the *match* is successful. Of importance is to note that "match" here means matching in the combinatory setting, that is *first-order matching*, which is much simpler than in λP (more details towards the end of this section). For example, $K_{S_y} x S_y$ is a redex, but $K_{S_y} x K_y$ is not. Note that the behaviour of the standard *CL*-combinators K and S is given by that of K_P and S_P, resp., when P is any variable. For this reason, we usually abbreviate K_x, for any variable x, as K (and analogously for S).

Combinators of the form K_P and S_P alone are not expressive enough to model reduction in λP for they lack the ability to pull applications apart. Since the pattern P in $\lambda P.M$ will be translated to an application in CL_P (except for the case in which P is a variable) we need to be able to define functions which expect arguments of the form MN and take them apart, returning terms like M, N and $S(KN)M$. For instance, in order to translate $\lambda(\lambda x.y).y$, we need a term which can access the variable y from the translation of $\lambda x.y$, which is Ky. To that effect, we define the combinators Π^1_{PQ} and Π^2_{PQ}, known as **projectors**. $\Pi^1_{PQ}(MN)$ reduces to M when M matches P and N matches Q. Analogously, $\Pi^2_{PQ}(MN)$ reduces to N under the same conditions.

Some examples of the translation we shall introduce in Section 4 follow. λ-terms translate to *CL*-terms as per the original translation. Abstractions with non-variable patterns translate to CL_P-terms where the head combinator (the leftmost combinator) is decorated with a pattern. For example, $\lambda(\lambda x.x).y$ translates to $K_{SKK}y$, and $\lambda(\lambda x.y).y$ translates to $S_{Ky}(K(SKK))\Pi^2_{Ky}$.

Consider the λP-term $(\lambda(\lambda x.x).y)(\lambda z.z)$, which matches $\lambda z.z$ against the pattern $\lambda x.x$ and, since the match is successful, reduces in one step to y. This term can be translated as $K_{SKK}y(SKK)$. The subscript SKK in K_{SKK} is a pattern (it is, in fact, the translation of the pattern $\lambda x.x$) which must be matched by the second argument of K_{SKK}, namely SKK, the translation of $\lambda z.z$. In this case matching is successful, and thus the translated term reduces to y, just like the original λP-term.

More complex terms may require more steps to reduce. For instance, $(\lambda(\lambda x.y).y)(\lambda w.z)$, which in λP reduces to z in one step, is translated to $S_{Ky}(K(SKK))\Pi^2_{Ky}(Kz)$. The latter requires several steps to reach a normal form:

$$\begin{aligned} S_{Ky}(K(SKK))\Pi^2_{Ky}(Kz) &\to K(SKK)(Kz)(\Pi^2_{Ky}(Kz)) \\ &\to SKK(\Pi^2_{Ky}(Kz)) \\ &\to K(\Pi^2_{Ky}(Kz))(K(\Pi^2_{Ky}(Kz))) \\ &\to \Pi^2_{Ky}(Kz) \to z \end{aligned}$$

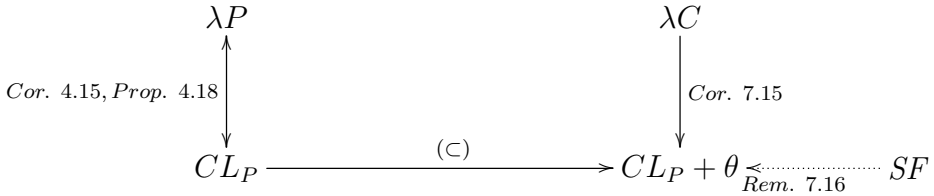
Matching failure is also carried over to the translation: $(\lambda(\lambda x.x).y)(\lambda w.z)$ translates to $K_{SKK}y(Kz)$, which also contains a matching failure, since Kz does not match the pattern SKK.

While at first sight it may seem as if having pattern matching requires substitutions to be computed (after all, a term M matches a pattern P if and only if there is a

substitution σ such that $M = P^\sigma$), actually computing a substitution is not necessary in our setting. All the matching algorithm has to do is decompose the pattern and the argument and check that, wherever there is a combinator in the pattern, the same combinator is present (in the same position) in the argument. Variables in patterns are automatically matched by any argument.

Another reason why matching is easier to compute in a combinator-based setting is the lack of binders. For example, in λP, the term $\lambda x.x$ does not match the pattern $\lambda x.y$, since there is no substitution which applied to $\lambda x.y$ returns $\lambda x.x$ (the convention set in [30] does not allow variable capture). However, this means that a matching algorithm would have to explicitly check that a variable not occur free in a term. On the other hand, if we translate the $\lambda x.y$ and $\lambda x.x$ into our combinatory logic system, we obtain Ky and SKK, and now matching failure is immediate from the fact that SK does not match K.

The following depicts, in a bird's eye view, how CL_P and variations relate to λP and its variations and also to SF (discussed below). All referred results are developed in the respective sections:



λP can be translated into CL_P while preserving (weak) reduction (Corollary 4.15) and the same holds in the reverse sense (Proposition 4.18). λC [25] is a variation of λP in which patterns are taken to be algebraic terms and also in which multiple pattern matching is allowed:

$$(\lambda P_1.M_1 | \ldots | \lambda P_k.M_k)P_i^\sigma \to M_i^\sigma$$

Constructors may be added to CL_P together with a new set of rules ($CL_P + \theta$) and the resulting system serves as target of a reduction preserving translation from λC. The reverse direction should hold as well (the results we develop have not required us to do so though). Although CL_P is a first-order term rewriting system composed of a finite number of rule schemas, an infinite number of rules result from instantiating these rule schemas. Finally, we briefly comment on SF [21], a combinatory logic similar in spirit to ours (see Section 8 for an in-depth comparison, and Section 7.3 for a partial translation from SF to $CL_P + \theta$ with additional patterns and generalized pattern matching).

We provide a recursive method to perform translations between the original language and our extension of CL. We will also see that it is possible to represent our calculus with a Term Rewriting System (TRS) with an infinite number of rules which are, however, captured by a finite number of schemas.

Our presentation develops a notion of CL for the λP-calculus, and then compares its basic properties with those of CL for the λ-calculus. We have found that most of them are well-preserved, confirming the adequacy of this extension. Moreover, we will see that pattern matching is preserved by the translation of λP-terms to our calculus. As a first order system, our calculus can both break (look inside) applications and encode the behavior of a higher order calculus while remaining simpler than a Combinatory Reduction System (CRS).

We propose some extensions and variants, including the introduction of constructors, new combinators, different sets of patterns, and a generalization of the matching mechanism which allows - among other things - the capture of certain forms of structural polymorphism.

We are also interested in formulating an adequate type system for our main calculus, whose filtering power is improved (with respect to the existing type systems for CL) by the presence of patterns. For example, while the terms KxS and K_KxK are typable, K_KxS is not - because S cannot be assigned the same type as the expected argument K.

It has been of fundamental importance to develop our proposal with the base of a formalism which can serve as a test-bed, having an adequate pattern-handling mechanism (see the preliminaries). That is why we have chosen the λP system: it similar to λ-calculus, with minimal extensions to represent patterns. While other calculi could be adapted accordingly, we have chosen λP as a starting point, as it is the simplest formulation, and all the other known pattern calculi are in some sense generalizations or variations of this calculus, based essentially on the same principles.

1.1 Related work

The original development of CL was initiated by Schönfinkel in [27]. Curry, Feys, Hindley and Seldin have studied it thoroughly [10, 11, 12, 13]. Gabbay and de Queiroz [16] have proposed celebrated approaches for restricting CL from the proof-theoretic standpoint. A good treatment of the topic can be found in [5] and, more extensively, in [28].

The notion of pattern matching has been present in functional programming languages during the last two decades. From the theoretical standpoint, many formulations are found in various published works. Based perhaps on the pioneering ideas of Peyton Jones [26], as well as the emergence of modern functional languages (Haskell, ML/CaML, Miranda, Clean), the λ-calculus with patterns [30] was proposed. This calculus represents a minimal and natural extension of the classical λ-calculus, including a notion of (instant and implicit) pattern matching as a part of the β-reduction rule. In [25] the system is revisited and compared with other formalisms. The λC-calculus, a variant of λP, is introduced in [25] as a means to handle matching with multiple

patterns. The ρ-calculus [9] is a very general formalism in which the pattern matching process is delayed (represented in an explicit way with rules that handle it sequentially). The Pure Pattern Type Systems [32, 31] are an extension of the Pure Type Systems [6] in which patterns are incorporated, defining in this way a hierarchy of typed calculi in a very general manner. The λ-calculus with constructors [1, 2] combines constructors with functions, and allows both sorts to receive the same treatment, with rules having the same behavior. The Pure Pattern Calculus [19, 22, 23] is a formalism in which patterns are first class citizens: it is possible to use a pattern as argument of a function and it is possible to return a pattern as function's result. It also allows the possibility of reduction inside the patterns themselves. This system is currently considered one of the most flexible, according to its representative power. In [17], a general logical framework is presented which allows the definition of different λ-calculi with their respective type systems, based on constraints which can be seen as generalizations of pattern-matching. Other logical frameworks are introduced as instances of the former, including a pattern logical framework. In [15], the author introduces the idea of "inverse combinators", which can match arguments against predefined structures composed by variables. The article focuses mainly on the equational point of view (logical derivations in substructural logics), rather than the rewriting point of view, and leaves out the K combinator, as it is not reversible. Finally, in [21], a combinatory calculus with the capability to decompose arguments is introduced; this feature, which relies on conditional rewriting rules, is used for defining different behaviors depending on the structures of the arguments.

A first version of our main calculus is presented by the authors in [3]. We believe that our contribution can be a starting point, and we are convinced that the current approach can be refined and/or improved in forthcoming works. All the formalisms mentioned above introduce languages with rules which handle patterns in different ways, thus they could be appropriate candidates for studying the notions we have developed for λP.

1.2 Structure of the paper

We start in Section 2 with a detailed overview of λP. We then introduce the CL_P rewriting system in Section 3. Section 4 treats the translation procedure for representing λP-terms and simulating the use of abstractions. Section 5 presents the equational theory associated to CL_P, and briefly discusses extensionality. In Section 6 we introduce a type system for CL_P, for which we prove Type Preservation and Strong Normalisation, and consider the addition of constructors to the calculus, keeping the same spirit. Some variants of the system are presented in Section 7; particularly, in Section 7.2, we present a way to define variants of the calculus with different notions of pattern-matching without losing confluence. Then, in Section 8, we discuss other pattern calculi and how CL_P relates to them. Finally, in Section 9, we conclude and discuss some avenues for

future work.

2 Preliminaries

We shall assume familiarity with the untyped and simply-typed λ-calculus, and SK-combinatory logic [5, 6, 10, 11, 12, 13, 27].

2.1 The λP-calculus

We will survey the λ-calculus with patterns (λP).

We recall the set of terms of the λP-calculus, the λP-terms, from the Introduction:

$$M, N, P ::= x \mid MN \mid \lambda P.M$$

In the abstraction $\lambda P.M$, P is the *pattern* and M the *body*. Applications are left-associative, as per the usual convention.

Definition 2.1 (Free variables of λP-terms) *Free variables of terms (extended as expected to sets of terms) are defined as follows:*

$$\begin{array}{lcl} \mathsf{FV}(x) & \triangleq & \{x\} \\ \mathsf{FV}(MN) & \triangleq & \mathsf{FV}(M) \cup \mathsf{FV}(N) \\ \mathsf{FV}(\lambda P.M) & \triangleq & \mathsf{FV}(M) - \mathsf{FV}(P) \end{array}$$

Definition 2.2 (Simultaneous substitution over λP-terms) *Let $\mathcal{X}$ be the set of λP-variables. A **substitution** in λP is a function σ from variables to λP-terms such that $\sigma(x) \neq x$ for only finitely many $x \in \mathcal{X}$. The (finite) set of variables that $\sigma(x)$ does not map to themselves is called the **domain** of σ: $\mathsf{dom}(\sigma) \triangleq \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$. If $\mathsf{dom}(\sigma) = \{x_1, \ldots, x_n\}$, then we write σ as*

$$\sigma = \{x_1 \leftarrow \sigma(x_1), \ldots, x_n \leftarrow \sigma(x_n)\}$$

*The **restriction** of σ to S, denoted $\sigma|_S$, is defined as: $\sigma|_S \triangleq \{x \leftarrow M \mid x \leftarrow M \in \sigma \wedge x \in S\}$. A substitution σ is **idempotent** $\sigma(\sigma(x)) = \sigma(x)$ if for every $x \in \mathsf{dom}(\sigma)$.*

The application of a substitution $\sigma = \{x_i \leftarrow N_i\}_{i=1,\ldots,n}$ to a term M, denoted $\sigma(M)$ or M^σ, is defined as follows:

$$\begin{array}{lcll} \sigma(x) & \triangleq & N_i, & \text{if } x = x_i \\ \sigma(x) & \triangleq & x, & \text{if } x \neq x_1, \ldots, x_n \\ \sigma(MN) & \triangleq & \sigma(M)\sigma(N) & \\ \sigma(\lambda P.M) & \triangleq & \lambda P.\sigma(M) & \end{array}$$

We assume the expected **free variable convention** over σ: $(\mathsf{dom}(\sigma) \cup \mathsf{FV}(\mathsf{ran}(\sigma))) \cap \mathsf{FV}(P) = \emptyset$.

One-hole λP-contexts are λP-terms with a unique "hole" which can occur anywhere in the term except inside abstraction patterns:

$$C ::= \Box \mid MC \mid CM \mid \lambda P.C$$

Reduction is given by the β_P relation generalizing the original β-reduction as follows:

$$(\lambda P.M)P^\sigma \to_{\beta_P} M^\sigma$$

We use $M \to_{\beta_P} N$ to denote that M reduces in one β_P-step to N, that is: $M = C[(\lambda P.R)P^\sigma]$ and $N = C[R^\sigma]$ where C is some context. In this case $(\lambda P.R)P^\sigma$ is called a β_P-**redex** and R^σ the β_P-**reduct**. Note that reduction can occur on either side of an application, and only on the right side (body) of an abstraction. This calculus does not allow reduction inside the patterns.

As exemplified in the Introduction, if arbitrary terms were permitted as patterns, the calculus would not be confluent. Two restrictions have been defined in order to ensure confluence of the calculus [25]. We discuss first a more general condition, *RPC*, and then a syntax-driven restriction called RPC^+. The first requires the auxiliary notion of **simultaneous reduction** $\multimap\!\!\!\!\Rightarrow$ allowing to contract an arbitrary set of pairwise non-overlapping redexes simultaneously:

DEFINITION 2.3 (SIMULTANEOUS REDUCTION) *The relation $M \multimap\!\!\!\!\Rightarrow N$ is inductively defined as follows:*

$$\frac{}{M \multimap\!\!\!\!\Rightarrow M} \qquad \frac{M \multimap\!\!\!\!\Rightarrow M' \quad N \multimap\!\!\!\!\Rightarrow N'}{MN \multimap\!\!\!\!\Rightarrow M'N'} \qquad \frac{M \multimap\!\!\!\!\Rightarrow M'}{\lambda P.M \multimap\!\!\!\!\Rightarrow \lambda P.M'}$$

$$\frac{M \multimap\!\!\!\!\Rightarrow M' \quad N_1 \multimap\!\!\!\!\Rightarrow N_1' \quad \ldots \quad N_k \multimap\!\!\!\!\Rightarrow N_k'}{(\lambda P.M)(P\{x_1 \leftarrow N_1, \ldots x_k \leftarrow N_k\}) \multimap\!\!\!\!\Rightarrow \; M'\{x_1 \leftarrow N_1', \ldots x_k \leftarrow N_k'\}}$$

For instance, $I(I(IK)) \multimap\!\!\!\!\Rightarrow IK$ – with I defined as $\lambda x.x$ and K as $\lambda x.\lambda y.x$ – by simultaneously contracting the outer and inner redexes, however it is not the case that $IIK \multimap\!\!\!\!\Rightarrow K$ since the redex IK has been created by the first step. The following is the first of the two conditions ensuring confluence in λP:

DEFINITION 2.4 (RIGID PATTERN CONDITION) *A set of patterns satisfies the rigid pattern condition (RPC), if for any pattern P in it and for any substitution of terms $N_1, ..., N_n$ for the free variables $x_1, ..., x_n$ of P we have:*

$$P\{x_1 \leftarrow N_1, \ldots, x_n \leftarrow N_n\} \multimap\!\!\!\!\Rightarrow P' \Rightarrow P' = P\{x_1 \leftarrow N_1', \ldots, x_n \leftarrow N_n'\} \wedge N_i \multimap\!\!\!\!\Rightarrow N_i' (1 \leq i \leq n)$$

Theorem 6 in [25] β_P is confluent if all patterns satisfy *RPC*.

The second condition is as follows:

Definition 2.5 (RPC^+) *The set RPC^+ consists of all λ-terms which:*

1. *are linear: (free) variables occur at most once;*
2. *are in normal form: they contain no β-pattern redex; and*
3. *have no active variables: they have no subterms of the form xM with x free.*

For example, $\lambda x.x$, $\lambda x.y$ and $\lambda x.\lambda y.x$ satisfy RPC^+ (and can thus be used as patterns), but $\lambda(\lambda x.y).y$, $\lambda x.\lambda y.xy$ and $(\lambda y.x)z$ do not.

Theorem 7 in [25] The set RPC^+ satisfies *RPC* and thus yields a confluent calculus.

More details and an analysis of properties of this calculus can be found in [30, 25]. For every notion of reduction R to be mentioned we will use:

- $\to_R$ for one R-reduction step
- $\overset{=}{\to}_R$ for its reflexive closure
- $\overset{+}{\to}_R$ for its transitive closure
- $\twoheadrightarrow_R$ for its reflexive-transitive closure.
- $=_R$ for its equivalence (reflexive-symmetric-transitive) closure.

3 The CL_P-Calculus

The set of CL_P-terms is described by the grammar:

$$M, N ::= x \mid K_M \mid S_M \mid \Pi^1_{MN} \mid \Pi^2_{MN} \mid MN$$

where x ranges over a given countably infinite set of variables $\mathcal{X}$. Application is left-associative, as usual. K_M, S_M, Π^1_{MN} and Π^2_{MN} will be the **combinators**, or primitive functions of our calculus. The combinators K_M and S_M are pattern-decorated versions of the *CL* combinators. Π^1_{MN} and Π^2_{MN}, which will be called **projectors**, have been introduced in order to extract information from an application by means of decomposition. Although the grammar admits arbitrary terms as subindices in K, S, Π^1 and Π^2, they will shortly be circumscribed to a subset that we shall dub **patterns** (Def. 3.5). We will use the letters P and Q to refer to patterns.

The aim of the set of CL_P-terms is to mimic the 3 steps required for λP-reduction:

1. matching the pattern against the argument, which if successful
2. yields bindings of the free variables of the pattern to subterms of the argument,
3. which are then applied to the body.

In CL_P, the combinators of the form S_P and K_P serve for the third phase, the projectors for the second phase, and the subscripts for the first phase.

REMARK 3.1 *Note that we have an infinite number of combinators, one S_P, K_P, Π^1_{PQ}, Π^2_{PQ} for each possible P and Q. For example, the CL_P-terms K_x, K_{K_y} and K_{S_y} are all different combinators.*

DEFINITION 3.2 (FREE VARIABLES OF CL_P-TERMS) *Free variables are defined as follows:*

$$\begin{array}{lcl} \mathsf{FV}(x) & \triangleq & \{x\} \\ \mathsf{FV}(MN) & \triangleq & \mathsf{FV}(M) \cup \mathsf{FV}(N) \\ \mathsf{FV}(K_P) = \mathsf{FV}(S_P) = \mathsf{FV}(\Pi^1_{PQ}) = \mathsf{FV}(\Pi^2_{PQ}) & \triangleq & \emptyset \end{array}$$

In the sequel of this chapter we use **term** for CL_P-term where there is no ambiguety. We will also adopt the following conventions:

- All free variables are considered to be different from the names of the variables in the patterns, and the latter different from each other, even if their names clash. There are no bound variables in this calculus; the presence of a variable in a pattern only serves to indicate that any term is a valid match.[1]
- We consider that two patterns are equal if they differ in nothing but the names of their variables, and we will work modulo renaming of variables in patterns (Def. 3.6).

Variables in terms are used in the same way as in *CL*. Variables within patterns – present as subscripts – play a slightly different role, as they represent a subterm of the pattern which can be matched (Definition 3.3) by any given term. They have no correspondence with any variables in the term in which they are involved, and as such they do not act as binders. In other words, the same variable cannot appear as both a subterm and (part of) a subscript within a term.

DEFINITION 3.3 *Substitution over CL_P-terms and patterns is defined as follows:*

[1] An alternative would be to omit variables in patterns, replacing each variable by a $\star$. We do not use this approach in CL_P as we are interested in providing a TRS formulation.

$$\begin{array}{lcl} x^\sigma & \triangleq & \sigma(x) \\ K_P^\sigma & \triangleq & K_P \\ S_P^\sigma & \triangleq & S_P \end{array} \qquad \begin{array}{lcl} (\Pi^1_{QR})^\sigma & \triangleq & \Pi^1_{QR} \\ (\Pi^2_{QR})^\sigma & \triangleq & \Pi^2_{QR} \\ (MN)^\sigma & \triangleq & M^\sigma N^\sigma \end{array}$$

*A term M is an **instance** of a pattern P (also, M **matches** the pattern P) if $\exists \sigma. P^\sigma = M$.*

Remark 3.4 *Note that substitutions* do not affect *the subscripts of combinators: each combinator K_P, S_P, Π^1_{QR}, Π^2_{QR} is a constant for all patterns P, QR. Patterns are affected by substitutions when used as terms, but not when used as subscripts. For example, the term Kx will be affected by substitutions which affect the variable x, but the term Π^1_{Kx} will not.*

We denote the unification relation as $\stackrel{\circ}{=}$. For example, $P \stackrel{\circ}{=} M$ should be read as "P unifies with M". Similarly, $P \not\stackrel{\circ}{=} M$ should be read as "P does not unify with M". The purpose of our use of unification in the definition that follows is to ensure that no unwanted redexes are formed (that is, that terms which match a pattern do not reduce to terms which no longer match it).

Definition 3.5 (Patterns) *The set of **patterns** is defined as follows:*

$$\begin{array}{lll} P, Q, P_1, \cdots P_n & ::= x & \\ & \mid\ K_P P_1 \cdots P_n & \text{where } n < 2 \vee P_2 \not\stackrel{\circ}{=} P \\ & \mid\ S_P P_1 \cdots P_n & \text{where } n < 3 \vee P_3 \not\stackrel{\circ}{=} P \\ & \mid\ \Pi^1_{PQ} P_1 \cdots P_n & \text{where } n = 0 \vee P_1 \not\stackrel{\circ}{=} PQ \\ & \mid\ \Pi^2_{PQ} P_1 \cdots P_n & \text{where } n = 0 \vee P_1 \not\stackrel{\circ}{=} PQ \end{array}$$

with $n \geq 0$ and $\mathsf{FV}(P_i) \cap \mathsf{FV}(P_j) = \emptyset$ for all $1 \leq i, j \leq n$ s.t. $i \neq j$ (this means, by recursion, that all patterns are linear).

A pattern of the form $P_1 P_2$ is called an **application pattern**. We abbreviate K_x and S_x, as K and S, respectively. Examples of patterns are: K, S, $K_{K_{S_{K^x}}}\ SKK$, $S_S KK$, Π^1_{KS}, $\Pi^2_{K_K x}$. The terms $K_S SK$, $S_K KSS$, $\Pi^1_{KS} K$, $\Pi^2_{K_K x}(KS)$ are also patterns because, while they have the number of arguments required to form a redex (or more), they contain pattern-matching failures[2] which prevent them from unifying with the left-hand side of any reduction rules: more precisely, K does not unify with S, KS nor K_K. On the other hand, xy, yS, $K_S SS$, $S_K KSK$, $\Pi^1_{KS}(KS)$ and $\Pi^2_{K_K x}(K_K S)$ are not patterns, as they all have subterms which unify with the left-hand side of a reduction rule. And, of course, non-linear terms like $Sx(Kx)$ or Syy are not patterns either.

[2]Not unifying with the left-hand-side of a reduction rule by itself does not guarantee a matching failure, since a term which does not unify with another may reduce to a term which does. However, this is not possible for patterns, since the definition is recursive and $P_1, \ldots, P_n$ must be patterns too.

The patterns defined by this syntax are those which conform to a restriction named RPC^{++}, which will be explained in detail in Section 3.1.1, once the reduction rules have been introduced.

DEFINITION 3.6 (α-EQUIVALENCE OVER CL_P-TERMS) *The fact that variable names within a pattern are irrelevant induces an α-equality relation between patterns, which is defined as follows:*

$$\begin{array}{lll} x & =_\alpha & y \\ K_P & =_\alpha & K_Q, \quad \text{if } P =_\alpha Q \\ S_P & =_\alpha & S_Q, \quad \text{if } P =_\alpha Q \\ \Pi^1_{PQ} & =_\alpha & \Pi^1_{P'Q'}, \quad \text{if } P =_\alpha P' \text{ and } Q =_\alpha Q' \\ \Pi^2_{PQ} & =_\alpha & \Pi^2_{P'Q'}, \quad \text{if } P =_\alpha P' \text{ and } Q =_\alpha Q' \\ PQ & =_\alpha & P'Q', \quad \text{if } P =_\alpha P' \text{ and } Q =_\alpha Q' \\ P & \neq_\alpha & Q, \quad \text{in any other case.} \end{array}$$

Equality between arbitrary CL_P-terms is defined in a similar way, except that different variables are treated as different terms. Only the subscripts of combinators are subject to α-conversion.

DEFINITION 3.7 *We use the following function to measure the size of a term:*

$$\begin{array}{lll} |x| & \triangleq & 1 \\ |K_P| = |S_P| & \triangleq & 1 + |P|, \quad \text{for every pattern } P \\ |\Pi^1_{QR}| = |\Pi^2_{QR}| & \triangleq & 1 + |Q| + |R|, \quad \text{for every patterns } Q,\ R \text{ such that } QR \text{ is a pattern} \\ |MN| & \triangleq & |M| + |N| \end{array}$$

3.1 Reduction in CL_P

The aim of this subsection is to introduce a rewriting system based on the above combinators, which will simulate λP in the sense of combinatorial completeness, as well as mimic the pattern matching of this calculus.

DEFINITION 3.8 (W_P-REDUCTION) ***W_P-reduction*** *(denoted as $\to_{W_P}$) is defined as the following* TRS *over the signature given by the syntax presented at the beginning of Section 3:*

$$\begin{array}{lll} K_P x P & \to & x, \quad x \notin \mathsf{FV}(P) \\ S_P x y P & \to & xP(yP), \quad x, y \notin \mathsf{FV}(P) \\ \Pi^1_{PQ}(PQ) & \to & P \\ \Pi^2_{PQ}(PQ) & \to & Q \end{array}$$

where P and Q range over patterns, and the application PQ – where used – is also a pattern.

These rules are schematic, since P and Q range over an infinite set of patterns. We have, in fact, an infinite number of rules captured by a finite number of schemas. Before presenting some examples, three easily verifiable properties of reduction: reduction is closed over CL_P, it does not create new free variables and is well-defined over α-equivalence classes of terms.

Lemma 3.9 *Suppose M is a CL_P-term and $M \to_{W_P} N$. Then:*

1. $N \in CL_P$.
2. $\mathsf{FV}(N) \subseteq \mathsf{FV}(M)$.
3. *$M =_\alpha M'$ implies there exists a term N' such that $M' \to_{W_P} N'$ and $N =_\alpha N'$.*

Example 3.10 *We will now show some reduction and pattern-matching examples.*

- $K_P x^\sigma P^\sigma \to_{W_P} x^\sigma$ *(for any substitution σ).*
- $S_{Kx}\Pi^1_{Ky}\Pi^2_{KS}(KS) \to_{W_P} \Pi^1_{Ky}(KS)(\Pi^2_{KS}(KS)) \to_{W_P} K(\Pi^2_{KS}(KS)) \to_{W_P} KS$. *The substitutions used here are $\{x \leftarrow S\}$, $\{y \leftarrow S\}$ and $\emptyset$ respectively.*
- $S_{Kx}\Pi^1_{Ky}\Pi^2_{KS}(KS) \to_{W_P} \Pi^1_{Ky}(KS)(\Pi^2_{KS}(KS)) \to_{W_P} \Pi^1_{Ky}(KS)S \to_{W_P} KS$ *is another possible reduction path.*
- $\Pi^2_{S\Pi^1_{Kx}}(S\Pi^1_{Ky})(KS_K) \to_{W_P} \Pi^1_{Ky}(KS_K) \to_{W_P} K$. *No substitution is involved in the first step, since $S\Pi^1_{Kx}$ is the same as $S\Pi^1_{Ky}$ due to α-equivalence between the subscripts. In the second step, we use the substitution $\{y \leftarrow S_K\}$.*
- $\Pi^2_{K_K S}(KS)$ *does not reduce, as KS does not match $K_K S$.*
- $\Pi^2_{S\Pi^1_{Kx}}(S\Pi^1_{KS})$ *does not reduce, since Π^1_{KS} does not match Π^1_{Kx} and thus $S\Pi^1_{KS}$ does not match $S\Pi^1_{Kx}$. Note that, although KS matches Kx, the same does not hold for combinators which have these patterns as their subscripts. This is because substitutions do not affect subscripts.*

Note that $P^\sigma Q^\sigma$ is the same as $(PQ)^\sigma$ by definition. This means that for a term of the form $\Pi^1_P M$ to be a redex, both P and M must be applications[3], and M must be an instance of P. A term is said to be *active* if it is used as the left-hand side of an application. The reason why we have restricted the syntax of our terms so that Π^1 and Π^2 may only have an application as their pattern should now be clear: according to the rules, an active projector with a non-application pattern (i.e. a variable or combinator) would never execute (that is, the projector applied to an argument would not reduce).

[3]We do not allow terms like Π^1_x or Π^2_x, since their presence could easily break the confluence of the calculus. For example, the term $\Pi^1_x KKK$ would reduce to two distinct normal forms: KK and $\Pi^1_x K$.

PROPOSITION 3.11 *CL_P is an extension of CL.*

Proof.- There is a direct mapping from CL to CL_P: variables translate to themselves, and the combinators S and K translate as S_x and K_x respectively (this mapping is univocal modulo α-conversion). Since a variable can be matched by any term, the reduction rules for S_x and K_x behave in the same way as the reduction rules for S and K in CL. ■

3.1.1 The RPC^{++} restriction and confluence in CL_P

In order to prove confluence (Corollary 3.17) and just as in λP, it is necessary to impose restrictions over the patterns. Without them, confluence would not hold: for instance, the term $\Pi^1_{xy}(KSK)$ would reduce to two different normal forms: KS and $\Pi^1_{xy}S$. Following Van Oostrom's RPC^+ restriction for λP, we call this set of restrictions $\boldsymbol{RPC}^{++}$. The aim of RPC^{++}, just like RPC^+, is to define a syntax-based, easily verifiable set of restrictions that can guarantee the confluence of the calculus.

DEFINITION 3.12 *The set of* ***application subterms*** $\mathtt{AS}(M)$ *in a CL_P-term M is defined as follows:*

$$\begin{array}{lcll} \mathtt{AS}(x) & \triangleq & \emptyset, & \\ \mathtt{AS}(M) & \triangleq & \emptyset, & \text{if } M \text{ is a combinator} \\ \mathtt{AS}(MN) & \triangleq & \{MN\} \cup \mathtt{AS}(M) \cup \mathtt{AS}(N) & \end{array}$$

The set of ***application patterns*** $\mathtt{AP}(M)$ *in a term M is defined as follows:*

$$\begin{array}{lcl} \mathtt{AP}(x) & \triangleq & \emptyset \\ \mathtt{AP}(K_P) & \triangleq & \mathtt{AS}(P) \cup \mathtt{AP}(P) \\ \mathtt{AP}(S_P) & \triangleq & \mathtt{AS}(P) \cup \mathtt{AP}(P) \\ \mathtt{AP}(\Pi^1_{PQ}) & \triangleq & \mathtt{AS}(PQ) \cup \mathtt{AP}(P) \cup \mathtt{AP}(Q) \\ \mathtt{AP}(\Pi^2_{PQ}) & \triangleq & \mathtt{AS}(PQ) \cup \mathtt{AP}(P) \cup \mathtt{AP}(Q) \\ \mathtt{AP}(MN) & \triangleq & \mathtt{AP}(M) \cup \mathtt{AP}(N) \end{array}$$

DEFINITION 3.13 (RPC^{++}) *A term M satisfies RPC^{++} if every $N \in (\mathtt{AS}(M) \cup \mathtt{AP}(M))$:*

1. *is linear, i.e. no variable appears more than once,*
2. *has no active variables, i.e. no subterms of the form xN' with $x \in \mathcal{X}$,*
3. *does not unify[4] with the left-hand-side of a W_P-rule.*

[4]Remember we are working modulo renaming of variables in patterns, so the pattern Ky will unify with yS even though K does not unify with S.

We do not require anything of non-application patterns (except, of course, that their subscripts – when present – satisfy RPC^{++}). A variable, for example, will trivially satisfy the RPC^{++} restriction. The second and third conditions imply that application patterns will have no active variables (i.e. they will have no subterms of the form xM), and that they will be normal forms. These two results, along with linearity (our first condition), constitute a translation of RPC^{+}, minus the requirement for all patterns to be λ-terms, to CL_P.

Remark 3.14 *M is a pattern if and only if M satisfies RPC^{++}. This can be easily verified by looking at the syntax of the patterns in Definition 3.5 and the definition of RPC^{++}.*

Our patterns are also rigid in that an instance of a given pattern may only reduce to another instance of the same pattern (see Lemma 7.6).

We briefly show with examples that our restrictions are well-motivated: breaking any of the last two conditions can result in a calculus that is not even locally confluent. (See for example the derivations starting from the terms $K_{xy}S(KKz)$, $K_{Kxz}y(KSS)$ and $K_{\Pi^1_{KK}xy}S(\Pi^1_{KK}(KK)z)$).

While the use of non-linear patterns may not lead to two different normal forms, it does break the confluence of the calculus. See Klop's standard example [24, 25], and define D as the term $S(K(S(K(K_{Sxx}E))))S$, where E is some term chosen to indicate equality.

This is the most general definition we will use for the set of patterns for CL_P. It is also possible to work with proper subsets of this set, in order to avoid dealing with unification and multiple levels of subscripts. See Section 7 for further details.

Lemma 3.15 *W_P is orthogonal in CL_P.*

Proof.- Left-linearity of the rules is immediate: wherever variables appear explicitly in a rule schema, they are required to be fresh with respect to the pattern; and patterns are required to be linear by RPC^{++}. The absence of critical pairs is a consequence of the restrictions that require all patterns – and their applicative subterms – not to unify with the left-hand-side of a W_P-rule. Since each rule of the TRS unifies with a rule in the original formulation, and all patterns satisfy RPC^{++}, this implies that no instance of a pattern, nor any of its subterms, matches the left side of any rule. ■

Remark 3.16 *Note that orthogonality follows crucially from the fact that the patterns involved in the rules satisfy RPC^{++}.*

The following corollary states the confluence of the calculus for every set of patterns satisfying RPC^{++}. It follows from the fact that no new patterns appear upon reduction and the previous lemma.

COROLLARY 3.17 *Let Φ be any subset of the CL_P-patterns, and let $CL_P(\Phi)$ be the CL_P-calculus restricted to the terms whose patterns belong to Φ. Then, $CL_P(\Phi)$ is confluent.*

4 Translations between λP and CL_P

In order to prove that W_P-reduction represents an abstraction mechanism, we will define translations between the two systems and then show the relationship between their respective reduction relations.

4.1 Translation from λP to CL_P

DEFINITION 4.1 (FROM λP-TERMS TO COMBINATORS) *Let M be a λP-term. Its corresponding combinator, denoted M_{CL}, is defined as follows:*

$$\begin{array}{rcl} x_{CL} & \triangleq & x \\ (MN)_{CL} & \triangleq & M_{CL}N_{CL} \\ (\lambda M.N)_{CL} & \triangleq & \lambda^* M_{CL}.N_{CL} \end{array}$$

with λ^* defined recursively as follows:

$$\begin{array}{lcll} 1)\,\lambda^* x.x & \triangleq & SKK & \\ 2)\,\lambda^* P.M & \triangleq & K_P M, & \text{if } \mathsf{FV}(P) \cap \mathsf{FV}(M) = \emptyset \\ 3)\,\lambda^* PQ.x & \triangleq & S_{PQ}(K\lambda^* P.x)\Pi^1_{PQ}, & \text{if } x \in (\mathsf{FV}(P) - \mathsf{FV}(Q)) \\ 4)\,\lambda^* PQ.x & \triangleq & S_{PQ}(K\lambda^* Q.x)\Pi^2_{PQ}, & \text{if } x \in \mathsf{FV}(Q) \\ 5)\,\lambda^* P.MN & \triangleq & S_P(\lambda^* P.M)(\lambda^* P.N), & \text{if } \mathsf{FV}(P) \cap \mathsf{FV}(MN) \neq \emptyset \end{array}$$

This translation extends to substitutions σ in λP as expected: $\sigma_{CL} \triangleq \{x \leftarrow M_{CL} \mid x \leftarrow M \in \sigma\}$.

Rules 1, 2 and 5 are inspired in the original translation from λ-calculus to CL. Rules 3 and 4 emerge from the necessity of decomposing application patterns. An abstraction of the form $\lambda PQ.x$, whose pattern expects an application, should be transformed into a term which executes projections until the location of x is found (either inside P or inside Q). After projecting, say, to the left (if $x \in \mathsf{FV}(P)$), the CL_P-derivation continues by letting the projected argument match with the pattern P: this is done by translating the term $\lambda P.x$ recursively. A term of the form $\lambda PQ.x$ should finally locate the corresponding instance of x inside the matching argument. Thus, the resulting translation will be a composition of S's, K's and projectors.

The fact that the function is well defined can be derived from a simple observation (all recursive calculations of λ^* are carried out over smaller terms). Furthermore, it can

be proved with a straightforward case by case analysis that any term of the form $\lambda P.M$ fits the hypotheses of one – and only one – of these 5 rules.

Clauses 1 to 5 are general enough to handle the full syntax without the RPC^{++} restriction. On the other hand, restricting the domain to terms with patterns will resolve the apparent lack of symmetry of rules 3 and 4: since a free variable never appears more than once in a pattern, their conditions can be simplified to $x \in \mathsf{FV}(P)$ and $x \in \mathsf{FV}(Q)$ respectively. Otherwise, without the restriction, one can always choose either case and the result will behave in the same way.

Remark 4.2 *The following rule may be added as a shortcut to optimize reductions (and reduce the size of the translated term):*

$$6)\ \lambda^* x.Mx \quad \triangleq \quad M, \quad \textit{if } x \notin \mathsf{FV}(M)$$

Clause 6 is optional, since all terms that can be translated with this rule can also be translated via rule 5, but it can greatly reduce the amount of reduction steps required to reach a normal form (whenever one exists). One may prove that the resulting terms are functionally equivalent, by showing that:

$$S_x(\lambda^* x.M)(\lambda^* x.x)N \quad \twoheadrightarrow_{W_P} MN, \quad \textit{if } x \notin \mathsf{FV}(M)$$

Indeed:

$$\begin{array}{lll} S_x(\lambda^* x.M)(\lambda^* x.x)N & =_{2)} & S_x(K_x M)(\lambda^* x.x)N \\ & =_{1)} & S_x(K_x M)(SKK)N \\ & = & S(KM)(SKK)N \\ & \to_{W_P} & (KMN)(SKKN) \\ & \to_{W_P} & M(SKKN) \\ & \twoheadrightarrow_{W_P} & MN \end{array}$$

Nevertheless, keeping this rule would result in a non-deterministic definition, unless the clauses are followed in a prescribed order by verifying the conditions of rule 6 before attempting to apply rule 5.

Just as in CL, it can be easily proved that $SKKN \twoheadrightarrow_{W_P} N$ for every term N. For this reason, we will allow the term SKK to be abbreviated as I to represent the identity function. Note that this only makes sense when the patterns involved are variables, otherwise pattern matching may fail. More generally, in CL_P we have for any pattern P:

$$S_P K K P^\sigma \to_{W_P} K P^\sigma (K P^\sigma) \to_{W_P} P^\sigma$$

The expression I_P is used to denote the term $S_P K K$.

REMARK 4.3 *The definition of abstraction λ^* extends the one for CL, i.e. for $M \in CL$ and $x \in \mathcal{X}$, $\lambda^* x.M$ coincides with the classical notion.*

REMARK 4.4 *While it is true that every abstraction in λP fits the hypotheses of one of the λ^* rules, terms which do not satisfy the RPC^+ restriction may translate to terms with ill-formed patterns. For example:*

$$(\lambda xx.x)_{CL} = \lambda^* xx.x =_{4)} S(K\lambda^* x.x)\Pi^2_{xx} =_{1)} S(KI)\Pi^2_{xx}$$

*but xx is **not** a pattern in CL_P.*

LEMMA 4.5 *If P is a CL_P-pattern, then so is $\lambda^* x.P$.*

Proof.- By induction on P.

- If $P = x$, then $\lambda^* x.P = SKK$, which is a CL_P-pattern.
- If $x \notin \mathsf{FV}(P)$, then $\lambda^* x.P = KP$, also a CL_P-pattern.
- If $P \neq x$ and $x \in \mathsf{FV}(P)$, then P must be an application P_1P_2 (with P_1 and P_2 patterns) and $\lambda^* x.P = S(\lambda^* x.P_1)(\lambda^* x.P_2)$. By IH, both $\lambda^* x.P_1$ and $\lambda^* x.P_2$ are CL_P-patterns. Therefore, so is $S(\lambda^* x.P_1)(\lambda^* x.P_2)$.

■

PROPOSITION 4.6 *Every λP-pattern which satisfies RPC^+ (Def. 2.5) translates into a CL_P-pattern.*

Proof.- By induction on the pattern. Keep in mind that, since it satisfies RPC^+, it must be a λ-term, and be either a variable or an abstraction of the form $\lambda x.P$ with P a pattern satisfying RPC^+(an application would contain either an active variable or a redex).

- If the pattern is a variable, then its translation is also a variable, which is a CL_P-pattern.
- If it is an abstraction $\lambda x.P$, then its translation is $\lambda^* x.P_{CL}$. Since P satisfies RPC^+, then by IH P_{CL} is a CL_P-pattern. Thus, by Lemma 4.5, so is $\lambda^* x.P_{CL}$.

■

On the other hand, some λP-terms which do **not** satisfy RPC^+ can still be translated to CL_P-patterns. Consider for example the term $(\lambda x.\lambda y.x)w$. This term does not satisfy RPC^+, as it is not a normal form. However, when we translate it to CL_P (using rules 2 and 6 of λ^*), we obtain the term $K_y w$, which satisfies RPC^{++}. In this sense, we can

say that the RPC^{++} restriction is more general than RPC^{+} (it allows a strictly larger set of patterns).

Even without rule 6, there are CL_P-patterns which are not a direct CL_P translation of any λP-patterns. For example, there is no λP-term M such that $M_{CL} = \Pi^1_{KS}$ (it is immediate from the definition of $-_{CL}$ that the result of this translation will never be a single projector), and yet the term Π^1_{KS} satisfies RPC^{++}.

Another alternative is to replace rule 1 by the more general rule:

$$1') \quad \lambda^* P.P \triangleq I_P = S_P K K, \quad \text{if } \mathsf{FV}(P) \neq \emptyset$$

and since I_P is the identity restricted to the set of terms matching P, the process will yield a more efficient translation. Its condition ensures it does not overlap with rule 2, but it will still overlap with rule 5, resulting in a non-deterministic system, just like the system that results of including rule 6. Naturally, precedences among the rules may be defined in order to regain determinism.

Remark 4.7 *λP-patterns which satisfy RPC^{+} will translate into a more restricted set of patterns, since they are λ-terms and therefore translate into CL-terms. The grammar for the patterns which result from such a translation is: $P ::= x \mid K \mid S \mid KP \mid SP \mid SPP$, maintaining linearity as usual. We will refer to this new set as RPC^{+}-patterns.*

4.1.1 Further results related to the translation

Lemma 4.8 ($-_{CL}$ preserves free variables) $\mathsf{FV}(M) = \mathsf{FV}(M_{CL})$.

Proof.- By induction on the definition of M_{CL} (free variables may only appear in x_{CL}, $(MN)_{CL}$ and rules 2 and 5 of λ^*). ■

Lemma 4.9 (Commutation of $-_{CL}$ and substitution) $(M^\sigma)_{CL} = (M_{CL})^{\sigma_{CL}}$.

Proof.- By induction on the size of M. Our IH is that $(M^\sigma)_{CL} = (M_{CL})^{\sigma_{CL}}$ for every λP substitution σ and for every λP-term M which is strictly smaller than the term we are analyzing. ■

Lemma 4.10 *If $M_{CL} = P^\sigma$, then there is a substitution σ' in λP such that $\sigma = \sigma'_{CL}$.*

Proof.- By induction on the pattern P, using the syntax for CL_P-patterns defined in Section 3.1.1. ■

Proposition 4.11 ($-_{CL}$ preserves pattern matching) *For every λP-term P such that P_{CL} is a CL_P-pattern, for every λP-term M: $(\exists \sigma \text{ s.t. } M = P^\sigma) \Leftrightarrow (\exists \sigma' \text{ s.t. } M_{CL} = P^{\sigma'}_{CL})$, where σ is a substitution in λP and σ' is a substitution in CL_P.*

Proof.- The $\Rightarrow$-direction is a direct consequence of Lemma 4.9 (take $\sigma' = \sigma_{CL}$). The $\Leftarrow$-direction is a consequence of Lemmas 4.10 and 4.9. Take any σ s.t. $\sigma' = \sigma_{CL}$ (by Lemma 4.10, such a σ exists). ■

4.2 Translation from CL_P to λP

Any CL_P-term M can be translated to a term M_λ in λP.

DEFINITION 4.12 (FROM COMBINATORS TO λP-TERMS) *Let M be a CL_P-term. Its corresponding λP term, denoted M_λ, is defined as follows:*

$$\begin{array}{lcll} x_\lambda & \triangleq & x & \\ (K_P)_\lambda & \triangleq & \lambda x.\lambda P_\lambda.x, & x \textit{ fresh} \\ (S_P)_\lambda & \triangleq & \lambda x.\lambda y.\lambda P_\lambda.xP_\lambda(yP_\lambda), & x \textit{ and } y \textit{ fresh} \\ (\Pi^1_{PQ})_\lambda & \triangleq & \lambda P_\lambda Q_\lambda.P_\lambda & \\ (\Pi^2_{PQ})_\lambda & \triangleq & \lambda P_\lambda Q_\lambda.Q_\lambda & \\ (MN)_\lambda & \triangleq & M_\lambda N_\lambda & \end{array}$$

This translation can be extended to substitutions in the same way as the previous one:

$$\sigma_\lambda \quad \triangleq \quad \{x \leftarrow M_\lambda \mid x \leftarrow M \in \sigma\}$$

Note that the domains of substitutions are preserved by both translations ($-_{CL}$ and $-_\lambda$).

It can be observed that, just like the previous translation, $-_\lambda$ preserves free variables: For every CL_P-term M, $\mathsf{FV}(M) = \mathsf{FV}(M_\lambda)$. This is immediate from the first 2 lines of the above definition, the definitions of free variables in CL_P and λP and the fact that none of the lines of the definition of $-_\lambda$ introduces nor eliminates free variables (the variables introduced in the third and fourth lines are bound).

REMARK 4.13 *$-_\lambda$ is not the inverse of $-_{CL}$. Expanding the definitions shows that $((\lambda x.x)_{CL})_\lambda$ is not the same as $\lambda x.x$, and that $((S_P)_\lambda)_{CL}$ is not the same as S_P. What does hold, nevertheless, is that $(M_{CL})_\lambda =_{\beta_P} M$ (or, if we consider rule 6 for λ^*, $(M_{CL})_\lambda =_{\beta_P\eta} M$). In fact, if we use the original definition of λ^* (without accelerator rules), we have that $(M_{CL})_\lambda \twoheadrightarrow_{\beta_P} M$. We prove this later as Proposition 4.20.*

4.3 Relationship between $\rightarrow_{W_P}$ and $\rightarrow_{\beta_P}$

We now analyze the connection between the W_P reduction in CL_P and the original β_P reduction in λP. We begin by proving that W_P can provide an abstraction mechanism (Corollary 4.15). We will also prove the following results:

- For all CL_P-terms M, N: if $M \rightarrow_{W_P} N$ then $M_\lambda \overset{+}{\rightarrow}_{\beta_P} N_\lambda$ (Prop. 4.18).
- For every λP-term M: $(M_{CL})_\lambda \twoheadrightarrow_{\beta_P} M$ (Prop. 4.20).

- For all CL_P-terms $M,\ N$: if $M =_{W_P} N$ then $M_\lambda =_{\beta_P} N_\lambda$ (Lem. 4.22).
- The $-_{CL}$ translation preserves higher-order unification and matching (Prop. 4.23).
- For all λP-terms $M,\ N$, if $M =_{\beta_P} N$ then $M_{CL} =_{W_P} N_{CL}$. (Cor. 4.24).

In order to prove *Abstraction Simulation* (Corollary 4.15), we need the following two lemmas:

1. $(M^\sigma)_{CL} = (M_{CL})^{\sigma_{CL}}$ for every λP-term M.
2. $(\lambda^* P.M)P^\sigma \twoheadrightarrow_{W_P} M^\sigma$ if $\mathsf{dom}(\sigma) \subseteq \mathsf{FV}(P)$.

The first of this lemmas has already been proved as Lemma 4.9. We now show the proof of the second lemma.

Lemma 4.14 $(\lambda^* P.M)P^\sigma \twoheadrightarrow_{W_P} M^\sigma$ *if* $\mathsf{dom}(\sigma) \subseteq \mathsf{FV}(P)$.

Proof.- By induction on the definition of $\lambda^* P.M$. ■

Corollary 4.15 (Abstraction Simulation) *For every CL_P-representable term M, CL_P-representable pattern P and λP substitution σ s.t.* $\mathsf{dom}(\sigma) \subseteq \mathsf{FV}(P)$*:*

$$((\lambda P.M)P^\sigma)_{CL} \twoheadrightarrow_{W_P} (M^\sigma)_{CL}.$$

Proof.- By Lemma 4.9, $((\lambda P.M)P^\sigma)_{CL} = (\lambda P.M)_{CL}(P_{CL})^{\sigma_{CL}} = (\lambda^* P_{CL}.M_{CL})(P_{CL})^{\sigma_{CL}}$, and $(M^\sigma)_{CL} = (M_{CL})^{\sigma_{CL}}$. The result holds by Lemma 4.14. ■

With this, we have finally proved that $((\lambda P.M)P^\sigma)_{CL} \twoheadrightarrow_{W_P} (M^\sigma)_{CL}$ whenever both sides of the $\twoheadrightarrow_{W_P}$ are well-formed CL_P-terms and the domain of σ is contained in FV(P). This result is an extension of its counterpart in *CL*: not only does $-_{CL}$ translate abstraction to λ^*, but it also applies the translation to the pattern.

Note that, just as in *CL*, the implication $M \to_{\beta_P} N \supset M_{CL} \twoheadrightarrow_{W_P} N_{CL}$ does not hold.

Other relevant results regarding the translations are listed below.

Lemma 4.16 $(M^\sigma)_\lambda = (M_\lambda)^{\sigma_\lambda}$.

Proof.- By induction on M. ■

Lemma 4.17 $(\sigma \circ \sigma')_\lambda = \sigma_\lambda \circ \sigma'_\lambda$.

Proof.- It is enough to prove that this holds whenever the compositions of the substitutions are applied to free variables (since only free variables will be affected). Whether the variable is contained in the domain of σ' or not, it can be easily proved that the result of applying either side of the equation to the variable is the same. ■

Reduction within CL_P functions as a weak reduction with respect to $\rightarrow_{\beta_P}$ (See section [8] for the original *CL*-analogue).

PROPOSITION 4.18 (WEAK REDUCTION) *If* $M \rightarrow_{W_P} N$ *then* $M_\lambda \overset{+}{\rightarrow}_{\beta_P} N_\lambda$

Proof.- By induction on the context of the reduction $M \rightarrow_{W_P} N$. In every case we make use of the result of Lemma 4.16 and the variable convention (that is, CL_P-patterns have no variables in common with other terms). Also keep in mind that the bound variables x and y introduced by the $-_\lambda$ translation are fresh variables, and that this translation does *not* introduce free variables that were not present in the original term. ■

COROLLARY 4.19 *If* $M_\lambda \in \mathrm{SN}_{\lambda P}$ *then* $M \in \mathrm{SN}_{CL_P}$.

Proof.- This is an immediate consequence of Proposition 4.18, as an infinite derivation in CL_P starting from M would result in an infinite derivation in λP starting from M_λ. ■

PROPOSITION 4.20 $(M_{CL})_\lambda \twoheadrightarrow_{\beta_P} M$.

Proof.- By induction on M. ■

REMARK 4.21 *The above proposition holds without the inclusion of accelerator rules. If we introduce rule 6, the* $\twoheadrightarrow_{\beta_P}$ *relation must be replaced by* $=_{\beta_P\eta}$ *since, for example,* $((\lambda x.yx)_{CL})_\lambda$ *will become* y*, which is* η*-equivalent but not* β_P*-equivalent to* $\lambda x.yx$.

While the translations we presented above can simulate abstractions correctly, there is still room for optimization. An alternate definition can be used in order to improve the efficiency (and, at the same time, reduce the size of the translated terms by decreasing the number of subindices) by avoiding the replication of subindices in each recursive call.

LEMMA 4.22 *If* $M =_{W_P} N$ *then* $M_\lambda =_{\beta_P} N_\lambda$.

Proof.- By confluence, there exists a term L such that $M \twoheadrightarrow_{W_P} L$ and $N \twoheadrightarrow_{W_P} L$. Then, by Proposition 4.18, $M_\lambda \twoheadrightarrow_{\beta_P} L_\lambda$ and $N_\lambda \twoheadrightarrow_{\beta_P} L_\lambda$. ■

PROPOSITION 4.23

1. *If σ unifies the equation $M_{CL} =_{W_P} N_{CL}$, then σ_λ unifies the equation $M =_{\beta_P} N$.*
2. *If σ is idempotent, then so is σ_λ.*
3. *If $M_{CL} =_{W_P} N^\sigma_{CL}$, then $M =_{\beta_P} N^{\sigma_\lambda}$ (higher-order matching is preserved).*

Proof.- We prove items 1 and 2, 3 being similar to 1.

1. We know that $(M_{CL})^\sigma =_{W_P} (N_{CL})^\sigma$. Then, by Lemmas 4.16 and 4.22,
$$\begin{array}{ll} ((M_{CL})_\lambda)^{\sigma_\lambda} & = \\ ((M_{CL})^\sigma)_\lambda & =_{\beta_P} \\ ((N_{CL})^\sigma)_\lambda & = \\ ((N_{CL})_\lambda)^{\sigma_\lambda}. & \end{array}$$
By Proposition 4.20, $(M_{CL})_\lambda \twoheadrightarrow_{\beta_P} M$ and $(N_{CL})_\lambda \twoheadrightarrow_{\beta_P} N$. This means that $((M_{CL})_\lambda)^{\sigma_\lambda} \twoheadrightarrow_{\beta_P} M^{\sigma_\lambda}$ and $((N_{CL})_\lambda)^{\sigma_\lambda} \twoheadrightarrow_{\beta_P} N^{\sigma_\lambda}$. Therefore $M^{\sigma_\lambda} =_{\beta_P} N^{\sigma_\lambda}$.
2. Since $\sigma \circ \sigma = \sigma$, then by Lemma 4.17: $\sigma_\lambda \circ \sigma_\lambda = \sigma_\lambda$.

■

Corollary 4.24 *If $M =_{\beta_P} N$ then $M_{CL} =_{W_P} N_{CL}$.*

Proof.- By item 3 of Proposition 4.23, using the identity substitution as σ. ■

5 The CL_P-Theory

Recall that the λ-calculus as well as CL can be formulated as equational theories (see for example [5, 6, 4, 8] for details about equational theories). The theory associated to CL_P is defined as interpreting all rewriting rules by an infinite set of equalities.

$$\begin{array}{lcll} K_P x P & \doteq & x, & x \notin \mathsf{FV}(P) \\ S_P x y P & \doteq & xP(yP), & x, y \notin \mathsf{FV}(P) \\ \Pi^1_{PQ}(PQ) & \doteq & P & \\ \Pi^2_{PQ}(PQ) & \doteq & Q & \end{array}$$

where P and Q range over CL_P-patterns, and the application PQ – where used – is also a pattern.

With this interpretation, the terms M and N will be considered equal whenever $M =_{W_P} N$. The CL_P-theorems are the provable equalities which follow from the above clauses, closed under applicative congruence and substitution.

Definition 5.1 *A theory $\mathcal{T}_1$ is a **proper consistent extension** of a theory $\mathcal{T}_2$ if:*

- $\mathcal{T}_1$ *is consistent,*
- *all* $\mathcal{T}_2$*-theorems are also* $\mathcal{T}_1$*-theorems, and*
- *some* $\mathcal{T}_1$*-theorems are not* $\mathcal{T}_2$*-theorems.*

REMARK 5.2 *Restricting the language to any family of patterns satisfying* RPC^{++}*, the equational* CL_P*-theory is a proper consistent extension of the equational CL-theory.*

Being a proper extension is straightforward. Consistency follows by taking any two different normal forms (such as x *and* y*), then by confluence they will not be convertible.*

5.1 Extensional CL_P

There is the question of whether the CL_P-theory is maximally consistent or not (sometimes known as Post consistent or saturated)[5]. We will see that, as in CL, the answer is again negative. One way to extend the theory is by adding extensionality [5, 6].

Let us consider incorporating the following rewriting rule to the original formulation of W_P, to be called η (in which every pattern is a variable):

$$S(Kx)(SKK) \;\; \rightarrow \;\; x$$

It is clear that, unlike its counterpart in the λ-calculus, the above rule does not require a specific variable not to be free in M: this has already been taken care of in the translation $\lambda^* x.Mx = M$ (see the comment after translation rule 6 in subsection 4.1).

REMARK 5.3 *This extension can simulate the traditional* η*-rule. That is,*

$$(\lambda x.Mx)_{CL} \twoheadrightarrow_{W_P+\eta} M_{CL}.$$

We can see that if we use rule 6 of λ^**, then* $(\lambda x.Mx)_{CL} = \lambda^* x.M_{CL}x = M_{CL}$*, which reduces to* M_{CL} *in 0 steps. On the other hand, if we do not allow rule 6, then* $(\lambda x.Mx)_{CL} = \lambda^* x.M_{CL}x = S(KM_{CL})(SKK)$*, which reduces to* M_{CL} *in 1 step of the* η*-rule.*

Note that our definition of RPC^{++} needs to be updated so that the clause that states patterns must not unify with the left-hand-side of a W_P-rule reads "$W_P + \eta$" rather than just W_P. Otherwise, orthogonality would not be valid anymore, as there would be an infinite number of critical pairs:

$$\Pi^j_{SPQ}(S(KM)(SKK)) \quad j = 1, 2$$

[5]That is, without adding specific reduction rules involving new combinators, which is always a possibility, but it is not the focus of this section.

where P, Q are patterns such that, for some substitution σ, $P^\sigma = KM$ and $Q^\sigma = SKK$. In all these terms, both η and projection are applicable and it is not possible to close the diagram. Thus this system would not even be weakly confluent.

The problem originates from the fact that a pattern like Syz above has instances which may η-reduce in the root (because $S(Kx)(SKK)$ may match with it). Therefore, in order to have a sound extensionality in the calculus, a new restriction should be added to RPC^{++}. We discuss such a condition below.

We will say that the pattern P is **η-forbidden** if there is an application QR such that QR is a subterm of P and $QR \doteq S(Kx)(SKK)$. That is, if any subterm of P is an application which unifies with the left-hand-side of the η rule. Let us call this new restriction for the set of patterns RPC^{++}_η.

Remark 5.4 *If a set Φ of patterns satisfies RPC^{++}, then the set $\{P \in \Phi \mid P$ is not η-forbidden$\}$ satisfies RPC^{++}_η.*

Let us write CL_η for $CL + \eta$ extending the original formulation, and $CL_{P\eta}(\Phi)$ for the rewriting system consisting of $CL_P(\Phi) + \eta$, i.e. in which, as usual, all patterns belong to Φ.

Proposition 5.5 *For any set Φ of patterns satisfying RPC^{++}_η, the rewriting system $CL_{P\eta}(\Phi)$ is confluent.*

Proof.- The resulting system is left-linear and does not have critical pairs, hence it is orthogonal. ■

The $CL_{P\eta}$-theory extends CL_P with extensionality, in which the η-rule is interpreted as an equality. Then we have, as before:

Corollary 5.6 *Restricting the language to any family of patterns satisfying RPC^{++}_η, the equational theory $CL_{P\eta}(\Phi)$ is a proper consistent extension of both the CL_η theory and the $CL_P(\Phi)$ theory.*

Proof.-

- Proper: $S(K\Pi^1_{SK})(SKK) \doteq \Pi^1_{SK}$ is a theorem in $CL_{P\eta}(\Phi)$ but not in CL_η nor $CL_P(\Phi)$.
- Consistent: S and K are two different normal forms with respect to $CL_{P\eta}(\Phi)$.
- Extension: the set of axioms $CL_{P\eta}(\Phi)$ contains the axioms of both CL_η and $CL_P(\Phi)$.

■

$$\frac{x^A \in \Gamma}{\Gamma \vdash x\colon A}\,(\textsc{Var}) \quad \frac{\Gamma' \vdash P\colon A}{\Gamma \vdash K_P\colon B \to A \to B}\,(\mathrm{K})$$

$$\frac{\Gamma' \vdash P\colon A}{\Gamma \vdash S_P\colon (A \to B \to C) \to (A \to B) \to A \to C}\,(\mathrm{S})$$

$$\frac{\Gamma' \vdash P\colon A \to B \quad \Gamma' \vdash Q\colon A}{\Gamma \vdash \Pi^1_{PQ}\colon B \to A \to B}\,(\textsc{Pi-1}) \quad \frac{\Gamma' \vdash P\colon A \to B \quad \Gamma' \vdash Q\colon A}{\Gamma \vdash \Pi^2_{PQ}\colon B \to A}\,(\textsc{Pi-2})$$

$$\frac{\Gamma \vdash M\colon A \to B \quad \Gamma \vdash N\colon A}{\Gamma \vdash MN\colon B}\,(\to\text{-}\textsc{Elim})$$

Figure 1: Typing rules for CL_P

6 A Simple Type System for CL_P

We now describe a type system for CL_P, named $CL^{\to}_P$, and prove some of its salient properties. **Type expressions** are defined by following grammar:

$$A, B \;::=\; \alpha \mid A \to B$$

where α ranges over a non-empty set of type variables $\mathcal{A}$. As usual, arrows in types are right-associative.

We will work with the following definitions regarding contexts. A **context** is a finite set of variables with type decorations $\{x_1^{A_1}, \ldots, x_n^{A_n}\}$, with $x_i \neq x_j$ for $i \neq j$.The **domain** of Γ, denoted $\mathsf{dom}(\Gamma)$, is defined as $\{x \mid \exists A(x^A \in \Gamma)\}$. Its **range**, denoted $\mathsf{ran}(\Gamma)$, is defined as $\{A \mid \exists x(x^A \in \Gamma)\}$. The intersection between two contexts $(\Gamma \cap \Gamma')$ is defined as: $\{x^A \mid x^A \in \Gamma \wedge x^A \in \Gamma'\}$. Similarly, the union of two contexts $(\Gamma \cup \Gamma')$ is defined as: $\{x^A \mid x^A \in \Gamma \vee x^A \in \Gamma'\}$, and is only defined if for every x, A and B: $x^A \in \Gamma \wedge x^B \in \Gamma' \supset A = B$. A substitution σ is **compatible** with a context Γ if for all types A, B and every $x \in \mathsf{dom}(\sigma)$: $(x^A \in \Gamma$ and $\Gamma \vdash x^\sigma\colon B \to A = B)$.

We also work with **type substitutions**, total functions from type variables to types. If $\delta(\alpha) = A$, then we often write $\alpha \leftarrow A \in \delta$. The domain of δ is $\{\alpha \mid \delta(\alpha) \neq \alpha\}$. Substitution over types (A^δ) and contexts is defined as:

$$\begin{array}{rcl} \alpha^\delta & \triangleq & \delta(\alpha) \\ (A \to B)^\delta & \triangleq & A^\delta \to B^\delta \\ \Gamma^\delta & \triangleq & \{x^{A^\delta} \mid x^A \in \Gamma\} \end{array}$$

The **typing rules** for $CL^{\to}_P$ are given in Fig. 1. Note that the patterns themselves must be typable for the term that contains them to have a type. We do not require

them to be typable in the same context as said term, but there has to be some context Γ' in which the pattern can be assigned the expected type. That is, the type that will be expected of the arguments which shall match it. This is because the pattern may have variables that will disappear once a substitution is applied. For example, the term $K_{SKx}S(SKK)$ is a ground term, which is typable in the empty context, in which its subterm SKK has type $A \to A$. However, the pattern SKx is not typable in the empty context because typing information is needed for the variable x. Hence the need for the context Γ', which assigns the required types to the variables in the pattern (an easy way to obtain one such Γ' is to extend Γ with the variables in the patterns, each with the type of the term that will match with it). Since patterns are linear and all their variables are fresh by convention, no conflicts may arise from extending the contexts in this manner.

Remark 6.1 *This type system is conservative with respect to the traditional one for simply typed CL: terms which are untypable in CL are also untypable in CL_P, and terms which have a type in CL are assigned the same type in CL_P. This follows from the observation that restricting the rules to terms with no projectors and only variables as their patterns, results in the original type system (all statements of the form $\Gamma' \vdash P\colon A$ trivially hold for $\Gamma' = \{P^A\}$ when P is a variable). It is not, however, conservative with respect to type inhabitation, as mentioned above.*

We now exhibit some examples.

Example 6.2 *We can prove that $\emptyset \vdash S_P KK : A \to A$, for any given type A and pattern P such that P can be assigned type A in some context.*

$$\dfrac{\dfrac{\dfrac{\Gamma' \vdash P\colon A}{\emptyset \vdash S_P\colon (A\to(C\to A)\to A)\to(A\to C\to A)\to A\to A}\,(\mathrm{S}) \quad \dfrac{\dfrac{}{\{x_1^{C\to A}\}\vdash x_1\colon C\to A}\,(\textsc{Var})}{\emptyset\vdash K_{x_1}\colon A\to(C\to A)\to A}\,(\mathrm{K})}{\emptyset\vdash S_P K\colon (A\to C\to A)\to A\to A}\,(\to\text{-}\textsc{Elim}) \quad \dfrac{\dfrac{}{\{x_2^{C}\}\vdash x_2\colon C}\,(\textsc{Var})}{\emptyset\vdash K_{x_2}\colon A\to C\to A}\,(\mathrm{K})}{\emptyset\vdash S_P KK\colon A\to A}\,(\to\text{-}\textsc{Elim})$$

Example 6.3 *The term $\Pi^2_{Sx}SK$ can be assigned the type $A \to B \to A$, as shown below. We split the derivation into three parts.*
Part 1:

$$\dfrac{\dfrac{\dfrac{}{\{x_1^A\}\vdash x_1\colon A}\,(\textsc{Var})}{\{x^{A\to B\to A}\}\vdash S_{x_1}\colon (A\to B\to A)\to(A\to B)\to A\to A}\,(\mathrm{S}) \quad \dfrac{}{\{x^{A\to B\to A}\}\vdash x\colon A\to B\to A}\,(\textsc{Var})}{\emptyset\vdash \Pi^2_{Sx}\colon((A\to B)\to A\to A)\to A\to B\to A}\,(\textsc{Pi-2})$$

Part 2:

$$\dfrac{\dfrac{\dfrac{}{\{x_2^A\} \vdash x_2 \colon A}\,(\textsc{Var})}{\{x^{A\to B\to A}\} \vdash S_{x_2} \colon (A \to B \to A) \to (A \to B) \to A \to A}\,(\mathrm{S}) \qquad \dfrac{\dfrac{}{\{x_3^B\} \vdash x_3 \colon B}\,(\textsc{Var})}{\emptyset \vdash K_{x_3} \colon A \to B \to A}\,(\mathrm{K})}{\emptyset \vdash SK \colon (A \to B) \to A \to A}\,(\to\text{-}\textsc{Elim})$$

Part 3:

$$\dfrac{\dfrac{\dots}{\emptyset \vdash \Pi^2_{Sx} \colon ((A \to B) \to A \to A) \to A \to B \to A}\,(\textsc{Pi-2}) \qquad \dfrac{\dots}{\emptyset \vdash SK \colon (A \to B) \to A \to A}\,(\to\text{-}\textsc{Elim})}{\emptyset \vdash \Pi^2_{Sx}(SK) \colon A \to B \to A}\,(\to\text{-}\textsc{Elim})$$

Note that these typing rules allow us to prove that $\{x^A\} \vdash K_K x K \colon A$. However, the term $K_K x S$ is untypable: it cannot be assigned a type in any context. This is because K_K can only be assigned types of the form $C \to (A \to B \to A) \to C$ for some types A, B and C, while S cannot have a type of the form $A \to B \to A$ for any types A and B. This is an example of how (mismatched) patterns may affect the typability of a term: if we removed the subscripts, the term KxS would, of course, be typable.

REMARK 6.4 *Note that every type is inhabited (for example, the term $\Pi^2_{Kx}(SKK)$ has any given type in the empty context). While this result makes it impossible to establish a Curry-Howard isomorphism with a consistent logic, it does not invalidate the use of this $CL^{\to}_P$ for typing purposes, as it still satisfies important properties like Subterm Typability, Weakening and Strengthening, Type Substitution, Type Preservation and even Strong Normalization of typable terms.*

6.1 Main Results

We now show some properties of $CL^{\to}_P$. The first is the Inversion Lemma whose proof is immediate from the syntax-driven nature of the typing rules.

LEMMA 6.5 (INVERSION LEMMA) *If $\Gamma \vdash M \colon D$ is derivable, then:*

- *if $M = x \in \mathcal{X}$ then $x^D \in \Gamma$.*
- *if $M = K_P$ for some pattern P then there exist A, B, Γ' s.t. $D = A \to B \to A$ and $\Gamma' \vdash P \colon B$.*
- *if $M = S_P$ for some pattern P then there exist A, B, C, Γ' s.t. $D = (A \to B \to C) \to (A \to B) \to A \to C$ and $\Gamma' \vdash P \colon A$.*
- *if $M = \Pi^1_{PQ}$ for some application pattern PQ then there exist A, B, Γ' s.t. $D = B \to A \to B$, $\Gamma' \vdash P \colon A \to B$ and $\Gamma' \vdash Q \colon A$.*

- *if* $M = \Pi^2_{PQ}$ *for some application pattern* PQ *then there exist* A, B, Γ' *s.t.* $D = B \to A$, $\Gamma' \vdash P\colon A \to B$ *and* $\Gamma' \vdash Q\colon A$.
- *if* $M = M_1 M_2$ *for some terms* M_1 *and* M_2 *then there exists* A *s.t.* $\Gamma \vdash M_1\colon A \to D$ *and* $\Gamma \vdash M_2\colon A$.

Corollary 6.6 *If a term is typable in an context* Γ*, so are all its subterms.*

Proof.- By structural induction on the original term, using the last item of the Inversion Lemma when the term is an application. ■

Corollary 6.7 *If* $\Gamma \vdash M : A$ *then* $\mathsf{FV}(M) \subseteq \mathsf{dom}(\Gamma)$.

Proof.- By structural induction on M. ■

The following lemma will allow for weakening and strengthening of type derivations, which will be needed in order to prove the Term Substitution property.

Lemma 6.8 (Weakening and Strengthening) *If* $\Gamma \vdash M : A$ *and* $\mathsf{FV}(M) \subseteq \mathsf{dom}(\Gamma \cap \Gamma')$*, then* $\Gamma' \vdash M : A$.

Proof.- By structural induction on M, using the Inversion Lemma (IL). ■

Corollary 6.9 *If* $\Gamma \vdash M : A$ *and* $\Gamma \subseteq \Gamma'$*, then* $\Gamma' \vdash M : A$.

Proof.- We know by Corollary 6.7 that if $\Gamma \vdash M : A$, then $\mathsf{FV}(M) \subseteq \mathsf{dom}(\Gamma)$. Also, because $\Gamma \subseteq \Gamma'$, $\Gamma \cap \Gamma' = \Gamma$ thus $\mathsf{dom}(\Gamma \cap \Gamma') = \mathsf{dom}(\Gamma)$. It is now immediate from Lemma 6.8 that $\Gamma' \vdash M\colon A$. ■

Lemma 6.10 *If* $\mathsf{FV}(M) = \emptyset$ *and* $\Gamma \vdash M\colon A$*, then* $\Gamma' \vdash M\colon A$.

Proof.- If $\mathsf{FV}(M) = \emptyset$, then $\mathsf{FV}(M) \subseteq \mathsf{dom}(\Gamma \cap \Gamma')$. Then, by Lemma 6.8, if $\Gamma \vdash M\colon A$ then $\Gamma' \vdash M\colon A$. ■

Lemma 6.11 *If* $\Gamma \vdash P^\sigma\colon A$ *then there is a context* Γ'*s.t.* $\Gamma' \vdash P\colon A$.

Proof.- By structural induction on P. ■

The following result states that if M can be typed in Γ extended with the adequate types for each variable in the domain of σ, then M^σ can be typed in Γ with the same type.

Lemma 6.12 (Term Substitution) *If* σ *is compatible with* Γ *and* $\Gamma \cup \{x_i^{A_i} \mid \exists M_i (x_i \leftarrow M_i \in \sigma \wedge \Gamma \vdash M_i\colon A_i)\} \vdash M\colon A$*, then* $\Gamma \vdash M^\sigma\colon A$.

Proof.- By induction on M. ■

Note that Lemma 6.12 generalizes the Term Substitution lemma for CL, in which the domain of the substitution σ is only one variable.

LEMMA 6.13 *If* $\Gamma \vdash M\colon A$ *then* $\Gamma^\delta \vdash M\colon A^\delta$.

Proof.- By induction on the derivation of $\Gamma \vdash M : A$. ■

Using the previous results, we have proved that our typing is preserved under reductions. In other words, our type system satisfies the following core property:

PROPOSITION 6.14 (TYPE PRESERVATION) *If* $M \to_{W_P} M'$ *and* $\Gamma \vdash M : A$ *then* $\Gamma \vdash M' : A$.

Proof.- By structural induction on M. ■

REMARK 6.15 *Notice that, whenever a pattern is involved in each case of the above proof, the information we obtain about the typability of this pattern from the Inversion Lemma turns out to be a consequence of other results we previously obtained (that is, for every pattern P, we reached the conclusion that $\Gamma' \vdash P\colon A$ in two different ways). This poses the question of whether the restrictions of the form $\Gamma' \vdash P\colon A$ on the rules are redundant. The answer is negative. In our proof of Type Preservation, we are always able to deduce $\Gamma' \vdash P\colon A$ by using Lemma 6.11 because we assume that the term M as a whole is typable, and thus know that P^σ can be given the type A in a context Γ. This is not the case if we want to type a combinator without passing it all its arguments. For example, $K_{S(SKK)(SKK)}$ is untypable, because its pattern $S(SKK)(SKK)$ cannot be typed in any context. If we removed the requirement for the pattern to be typable from the* K*-rule, the term $K_{S(SKK)(SKK)}$ would be typable, which is undesirable because $S(SKK)(SKK)$ has no typable instances and thus an application of the form $K_{S(SKK)(SKK)}M_1 \cdots M_n$ would never reduce (except, of course, inside each M_i).*

We will now introduce some definitions and lemmas in order to prove the strong normalization of typable terms. Our proof is based on Sørensen and Urzyczyn's proof for CL [28], which uses a simplified version of Tait's *computability* method [29].

Let SN denote the set of all strongly normalizable CL_P-terms (we will refer to the terms in this set as SN-terms). For each type B we define the standard notion of a set $[\![B]\!]$ of CL_P-terms *computable* in B:

$$\begin{aligned} [\![\alpha]\!] &\triangleq \text{SN} \\ [\![(A \to C)]\!] &\triangleq \{M \mid \forall N(N \in [\![A]\!] \supset MN \in [\![C]\!])\} \end{aligned}$$

The following four lemmas are necessary in order to prove the strong normalization result.

Lemma 6.16

(i) $[\![B]\!] \subseteq \mathrm{SN}$.

(ii) If $H_1, ..., H_k \in \mathrm{SN}$, *then* $xH_1...H_k \in [\![B]\!]$.

Proof.- By induction on B. ■

Lemma 6.17

1. *If* $LN(MN)H_1...H_k \in [\![B]\!]$, *then* $S_P LMNH_1...H_k \in [\![B]\!]$.
2. *If* $MH_1...H_k \in [\![B]\!]$, *then* $K_P MNH_1...H_k \in [\![B]\!]$.
3. *If* $(MN) \in \mathrm{SN}$ *and* $MH_1...H_k \in [\![B]\!]$, *then* $\Pi^1_{QR}(MN)H_1...H_k \in [\![B]\!]$.
4. *If* $(MN) \in \mathrm{SN}$ *and* $NH_1...H_k \in [\![B]\!]$, *then* $\Pi^2_{QR}(MN)H_1...H_k \in [\![B]\!]$.

Proof.- By induction on B, using Lemma 6.16. ■

Lemma 6.18 *If* M *is not an application*[6] *and* $H_1, ..., H_k$ *are* SN*-terms:* $\Pi^1_{QR}MH_1...H_k \in [\![B]\!]$ *and* $\Pi^2_{QR}MH_1...H_k \in [\![B]\!]$.

Proof.- By induction on B. ■

Lemma 6.19 *If* $\Gamma \vdash T : B$ *then* $T \in [\![B]\!]$.

Proof.- By structural induction on the term T, using the Inversion Lemma (IL). ■

Finally, we conclude with our main result of the section; it is an immediate consequence of Lemmas 6.16(i) and 6.19.

Proposition 6.20 (Strong Normalization) *Every typable* CL_P*-term is SN.*

Remark 6.21 *It would be interesting to develop a (dependent) type system capable of detecting matching failure. For example, one where* K_P *had type* $A \to B \to_P A$, *where the type "*$A \to_P B$*" means "function with domain in* A *and range in* B, *where the argument must match the pattern* P*" and "*$A \to B$*" is the same as "*$A \to_x B$*". This approach would prevent matching failure if* ($\to$-Elim) *requires the matching to be successful for the application to be well-typed. However, the difficulty lies in typing applications of the form* MN *with* M *of type* $A \to_P B$ *and in which* N *reduces to an instance of* P *but is not one yet.*

[6]That is, M is either a variable, a combinator, or a constructor if we include constructors in the calculus (see Section 6.2).

6.2 Modelling Data Structures

Constructors for modelling applicative data structures may be incorporated into CL_P. This is achieved by enriching CL_P with constants taken from some given set of constants $\mathcal{C}$. We illustrate with an example.

EXAMPLE 6.22 *If we are interested in supporting lists, we may assume constructors* $\mathtt{nil}$ *and* $\mathtt{cons}$ *belong to* $\mathcal{C}$*. A list with the elements* M_1 *and* M_2 *will be denoted by the term* $\mathtt{cons}\ M_1\ (\mathtt{cons}\ M_2\ \mathtt{nil})$*. Having constructors as patterns allows us to define terms which can only be applied to structures built with a certain constructor. For example, the term* $S_{\mathtt{cons}\,x\,y}(K\Pi^2_{\mathtt{cons}\,z})\Pi^1_{\mathtt{cons}\,v\,w}$ *will return the head of a non-empty list, and do nothing when applied to a term of any other form.*

In general, a data structure takes the form $CM_1 \cdots M_n$, with $C \in \mathcal{C}$, and will be a normal form whenever $M_1, \cdots M_n$ are normal forms. The set of patterns can also be extended with constructors: if C is a constructor and $P_1, \cdots P_n$ are patterns ($n \geq 0$), then $CP_1 \cdots P_n$ is a pattern. As expected, $\mathsf{FV}(C) = \emptyset$ and $C^\sigma = C$ for all $C \in \mathcal{C}$ and for every substitution σ. When translating between λP and CL_P, constructors translate to themselves, and the results we have obtained in 4.1.1 and 4.3 still hold.

The corresponding extension of the type system consists in adding a new rule to determine the type of each constructor. For example, we can extend our calculus with natural numbers by introducing the primitive type `Nat`, and the constructors `0` and `succ`. The associated typing rules would be the following:

$$\frac{}{\Gamma \vdash \mathtt{0} : \mathtt{Nat}}\ (\textsc{Zero}) \qquad \frac{}{\Gamma \vdash \mathtt{succ} : \mathtt{Nat} \to \mathtt{Nat}}\ (\textsc{Succ})$$

We can go one step further by allowing the use of parametric types, in order to achieve some measure of polymorphism. For example, we can define the type `List` A for any given type A, with $<>$ and `cons` as its constructors.

$$\frac{}{\Gamma \vdash <> : \mathtt{List}\ A}\ (\textsc{Nil}) \qquad \frac{}{\Gamma \vdash \mathtt{cons} : A \to \mathtt{List}\ A \to \mathtt{List}\ A}\ (\textsc{Cons})$$

This concept can be generalized to types with an arbitrary number of parameters (this is useful, for example, when defining tuples).

The properties mentioned in section 6.1 still hold when constructors are introduced. In the case of the Inversion Lemma, a new clause will be added for each new constructor. Since the addition of constructors does not introduce new reductions, Type Preservation and Strong Normalization are not affected by this extension (Lemma 6.16 must be extended with a new clause to deal with terms of the form $CH_1...H_k$, but these terms behave in the same way as those of the form $xH_1...H_k$, so the proof is analogous and, as a result, every constructor is computable in every type).

Remark 6.23 *The CL_P-calculus may be extended with other constants besides constructors, namely* functional constants. *These could be incorporated as syntactic sugar to facilitate data manipulation (just like I and I_P were previously defined as SKK and S_PKK respectively). For instance, the function* **pred** *may be introduced with the purpose of obtaining the predecessor of a natural number, and the functions* **head** *and* **tail** *can be used to observe the elements in a list. These functions can be defined as follows:*

$$\begin{array}{lcl} \mathtt{pred} & \triangleq & \Pi^2_{\mathtt{succ}\,x} \\ \mathtt{head} & \triangleq & S_{\mathtt{cons}\,x\,y}(K\Pi^2_{\mathtt{cons}\,z})\Pi^1_{\mathtt{cons}\,v\,w} \\ \mathtt{tail} & \triangleq & \Pi^2_{\mathtt{cons}\,x\,y} \end{array}$$

The definition of **head** *may seem complicated at first sight, but note that, when translated to λP, the normal form of the resulting term is $\lambda\mathtt{cons}\,x\,y.x$. Following the typing rules we have defined, it is easy to verify that $\Gamma \vdash \mathtt{pred}\colon \mathtt{Nat} \to \mathtt{Nat}$, $\Gamma \vdash \mathtt{head}\colon \mathtt{List}\,A \to A$ and $\Gamma \vdash \mathtt{tail}\colon \mathtt{List}\,A \to \mathtt{List}\,A$ for every context Γ and every type A.*

7 Defining Extensions and Variants

This section considers modifications of CL_P in which either the set of terms or reduction rules are modified in some way or another. In order to aid the reader in following these modifications, we classify them into two kinds loosely specified as follows:

- A **variant** of a calculus is a new calculus which is obtained from the former by making minor modifications to the set of terms and/or the reduction rules (for example, by imposing restrictions to the terms which are considered well-formed, removing a reduction rule or replacing it by a slightly different one). The modifications must be 'minor' in the sense that the connection to the original calculus is still evident.
- An **extension** is a particular form of variant in which nothing is removed or replaced; new terms and reduction rules may be added, maintaining the old ones.

A summary of the variants and extensions follows

CL_P with matching-safe patterns	Variant	Section 7.1.1
CL_P with generalized pattern matching	Extension	Section 7.2
CL_P with multiple matching	Extension	Section 7.3
CL_P with structural polymorphism	Extension	Section 7.3

We will now see some examples and related properties.

7.1 Possible Restrictions to the Set of Patterns

The RPC^{++} restriction allows for a wide variety of patterns, with unlimited length and depth. It was defined in this way in order to allow the user as much freedom as possible, but in some cases this freedom may not be necessary, and simpler sets of patterns may be used. We will now show some possibilities.

7.1.1 Matching-safe patterns

While the RPC^{++} restriction presented in Section 3.1.1 is enough to guarantee confluence, it is possible to impose further restrictions in order to avoid dealing with unification while checking the syntax of patterns. This is done by eliminating patterns which contain matching failures within them. In a setting where matching failures are unwanted, terms of the form $K_K xS$ would be considered undesirable, and thus we would not want them to be used as patterns, since they would only be matched by terms of that same unwanted form.

In this variant, the syntax of patterns is restricted to:

$$P, Q, P_1, \cdots P_n ::= \quad x \mid K_P \mid S_P \mid K_P P_1 \mid S_P P_1 \mid S_P P_1 P_2 \mid \Pi^1_{PQ} \mid \Pi^2_{PQ} \mid C P_1 \cdots P_n$$

with $n \geq 0$, $P'P$ a pattern, and maintaining linearity. Here C represents any constructor.

Using this set of patterns results in a confluent calculus, as it is a subset of the patterns that satisfy RPC^{++}. It also removes the hassle of having to use unification to determine whether a pattern is well-formed. We will refer to the patterns in this set as *matching-safe patterns*.

REMARK 7.1 *Every λP-pattern which satisfies RPC^{+} translates into a matching-safe CL_P-pattern: since RPC^{+} restricts patterns to original λ-terms, the result of the translation of a λP-pattern satisfying RPC^{+} to CL_P is a CL-term (all its subscripts are variables) and, by Remark 4.6, it is also a CL_P-pattern. As such, they must fit the syntax presented in Section 3.1.1, and they may not contain combinators whose subscripts do not unify with one of their arguments (because variables unify with everything). Thus, the only option left is for that argument not to be present.*

7.2 Parameterizing Pattern-Matching

So far we have presented a calculus which handles pattern-matching by means of substitutions, using the syntactic matching mechanisms of a TRS. In this section we explore a formulation of CL_P in which the pattern matching process is abstracted away and computed outside the formalism itself. The reduction rules of CL_P (cf. Definition 3.8) are replaced by the contextual closure of the following ones:

$$\begin{array}{lll} K_P MN & \to M & \Leftarrow \text{match}(P, N) \\ S_P M_1 M_2 N & \to M_1 N(M_2 N) & \Leftarrow \text{match}(P, N) \\ \Pi^1_{PQ}(MN) & \to M & \Leftarrow \text{match}(PQ, MN) \\ \Pi^2_{PQ}(MN) & \to N & \Leftarrow \text{match}(PQ, MN) \end{array}$$

The resulting system is a conditional TRS, where reduction depends on the matching predicate. The first argument of match must be a pattern, and the second, an arbitrary CL_P-term. The use of an external matching predicate will be referred to as **generalized pattern matching**. If we define $\text{match}(P, M) = (\exists \sigma$ substitution such that $M = P^\sigma)$, then the resulting reduction system will be $\to_{W_P}$.

A recursive formulation of the standard syntactic matching would be the following:

$$\begin{array}{lcll} \text{match}(P, P) & \triangleq & \texttt{True} & \\ \text{match}(x, M) & \triangleq & \texttt{True} & \text{if } x \in \mathcal{X} \\ \text{match}(P, M) & \triangleq & \texttt{False} & \text{if } P \text{ is a combinator or constructor and } M \neq P \\ \text{match}(PQ, MN) & \triangleq & \text{match}(P, M) \wedge \text{match}(Q, N) & \end{array}$$

Replacing this by a different matching predicate would yield a different reduction system. The matching predicate can be made as simple or as complex as desired, ranging from a simple syntactic verification to an elaborate algorithm. Confluence depends on the matching function. If, for example, we defined the matching function so that $\text{match}(P, M) \Leftrightarrow (P \in \mathcal{X} \vee M = PP)$, then the term $K_{KS}x(KS(KS))$ would have two different normal forms: x and $K_{KS}xS$. We will see that the new calculus will be confluent if the resulting **parallel reduction** is **coherent** (Def. 7.5).

7.2.1 Parallel Reduction and Coherence

Definition 7.2 *Given a reduction $\to$, we define its **parallel reduction** $\rightrightarrows$ inductively as follows:*

$$\frac{M \to N}{M \rightrightarrows N} \qquad \frac{}{M \rightrightarrows M} \qquad \frac{M \rightrightarrows M' \quad N \rightrightarrows N'}{MN \rightrightarrows M'N'}$$

The idea of the parallel reduction is to reduce all terms involved in an application simultaneously. Note that, unlike the simultaneous reduction in λP, the parallel reduction cannot contract nested redexes in one step. For example, $K(Kxy)z$ does not reduce to x in one step of $\rightrightarrows$.

Lemma 7.3 *For every pattern P, term Q, and substitution σ: if $P^\sigma \rightrightarrows Q$ then there is a σ' s.t. $Q = P^{\sigma'}$, where $\rightrightarrows$ is the parallel reduction for $\to_{W_P}$.*

Proof.- By structural induction on the pattern P. ■

LEMMA 7.4 *For every application pattern PQ and all substitutions σ, σ': if $(PQ)^\sigma \rightrightarrows (PQ)^{\sigma'}$, then $P^\sigma \rightrightarrows P^{\sigma'}$ and $Q^\sigma \rightrightarrows Q^{\sigma'}$.*

Proof.- PQ is an application pattern, which means it must be of the form $TM_1 \cdots M_n$, where $n \geq 1$, $M_1, \cdots M_n$ are patterns and T is either a combinator or a constructor. A simple induction on the number of arguments can prove that $P^\sigma = TM_1^\sigma \cdots M_{n-1}^\sigma \rightrightarrows TM_1^{\sigma'} \cdots M_{n-1}^{\sigma'} = P^{\sigma'}$ and $Q^\sigma = M_n^\sigma \rightrightarrows M_n^{\sigma'} = Q^{\sigma'}$, using Lemma 7.3 and the IH when $n > 1$. ■

DEFINITION 7.5 *A reduction $\to_R$ satisfies the **coherence** property if:*

1. *For every pattern P and all terms M, N: if* $\mathrm{match}(P, M)$ *and $M \to_R N$, then* $\mathrm{match}(P, N)$.
2. *For every application pattern PQ and all terms M, N, T: if* $\mathrm{match}(PQ, MN)$ *and $MN \to_R T$, then there are terms M', N' s.t. $T = M'N'$, $M \stackrel{=}{\to}_R M'$ and $N \stackrel{=}{\to}_R N'$.*

Intuitively, the first condition means that a term which matches a pattern can only reduce to terms which match the same pattern; while the second condition means that, if an application matches an application pattern, then the result of any one-step reduction of this application can be reached by reducing (in zero or one steps) each side of the application separately.

LEMMA 7.6 *The parallel reduction associated with $\to_{W_P}$ is coherent:*

1. *For every pattern P, and all CL_P-terms M, N: if* $\mathrm{match}(P, M)$ *and $M \rightrightarrows N$, then* $\mathrm{match}(P, N)$.
2. *For every application pattern PQ and all CL_P-terms M, N, T: if* $\mathrm{match}(PQ, MN)$ *and $MN \rightrightarrows T$, then there exist terms M', N' s.t. $T = M'N'$, $M \stackrel{=}{\rightrightarrows} M'$ and $N \stackrel{=}{\rightrightarrows} N'$.*

Proof.- Part 1 is a direct consequence of Lemma 7.3, as

$$\mathrm{match}(P, M) = (\exists \sigma \text{ substitution s.t. } M = P^\sigma)$$

(and analogously for N). Simply replace M by P^σ and N by $P^{\sigma'}$, with σ' the substitution obtained from Lemma 7.3.

Similarly, part 2 is a direct consequence of Lemma 7.4, replacing M by P^σ, N by Q^σ, M' by $P^{\sigma'}$, N' by $Q^{\sigma'}$, and $\stackrel{=}{\rightrightarrows}$ by $\rightrightarrows$ (since $\rightrightarrows$ is reflexive, it is equal to its reflexive closure). T will always be an application by Lemma 7.3, as only an application can match another application. ■

Corollary 7.7 *The RPC^{++} restriction ensures the coherence of the parallel reduction.*

Proof.- This is a direct consequence of Lemma 7.6. ■

We will say that a calculus is a **coherent variant** of the CL_P-calculus if its reduction rules fit the schema presented at the beginning of Section 7.2 and its parallel reduction is **coherent** (that is, it satisfies the coherence property). This concept was inspired by Van Oostrom's original *RPC* restriction (see Definition 2.4).

7.2.2 Confluence

We will now prove the confluence of any coherent variant of the CL_P-calculus (with or without constructors). We can no longer use the orthogonality result, as we have no guarantee that our variant can be formulated as an orthogonal TRS (or even an orthogonal Higher-order Rewriting System). Instead, we will use the parallel reduction technique. To this effect, we will use the following definition and lemma. A reduction $\rightarrow_R$ satisfies the **diamond** property (denoted as $\rightarrow_R \models \diamond$) if whenever $A \rightarrow_R B$ and $A \rightarrow_R C$, there is a D s.t. $B \rightarrow_R D$ and $C \rightarrow_R D$.

Lemma 7.8 *Let $\rightarrow_1$ be a binary relation; if there exists $\rightarrow_2$ s.t. $\rightarrow_1 \subseteq \rightarrow_2 \subseteq \twoheadrightarrow_1$ and $\rightarrow_2 \models \diamond$, then $\rightarrow_1$ is confluent.*

Proof.- See [4, 8]. ■

The following lemmas show that the parallel reduction satisfies the hypotheses of Lemma 7.8, using the reduction defined for our calculus ($\rightarrow$) as $\rightarrow_1$.

Lemma 7.9 $\rightarrow \subseteq \rightrightarrows \subseteq \twoheadrightarrow$

Proof.- The first inclusion is immediate from the first rule of the definition of $\rightrightarrows$.

The second inclusion derives from the fact that $\twoheadrightarrow$ is the reflexive transitive closure of our reduction $\rightarrow$. This means that:

1. $\rightarrow \subseteq \twoheadrightarrow$, which covers rule 1 of the definition of $\rightrightarrows$.
2. $\twoheadrightarrow$ is reflexive, which covers rule 2.
3. $\twoheadrightarrow$ is transitive. If $M \twoheadrightarrow M'$, $N \twoheadrightarrow N'$ and $MN \twoheadrightarrow M'N \twoheadrightarrow M'N'$, then $MN \twoheadrightarrow M'N'$. This covers rule 3 of the definition of $\rightrightarrows$. Therefore, $\rightrightarrows \subseteq \twoheadrightarrow$. ■

It can be proved that $\rightrightarrows$ satisfies the diamond property, using the fact that $\rightrightarrows$ is coherent.

LEMMA 7.10 *If $\rightarrow$ is the reduction associated to a coherent variant of CL_P, then $\rightrightarrows \models \diamond$.*

Proof.- In order to prove that $\rightrightarrows$ satisfies the diamond property, we will use the fact that $\rightrightarrows$ is coherent. We can prove that $M \rightrightarrows M_1 \wedge M \rightrightarrows M_2 \supset \exists M_3$ s.t. $M_1 \rightrightarrows M_3 \wedge M_2 \rightrightarrows M_3$ by induction on the reduction $M \rightrightarrows M_1$. ■

As a result of Lemmas 7.8, 7.9 and 7.10, every coherent variant of CL_P is confluent.

REMARK 7.11 *Note that when using this generalized notion of pattern matching, patterns are no longer restricted to those satisfying RPC^{++}. Any coherent variant will be confluent.*

7.3 Representing other pattern calculi

By introducing new terms and/or changing the set of allowed patterns we obtain different calculi, which can introduce new features to our calculus and represent different languages.

The λC-calculus. For example, we can translate λC into a variant of CL_P. Recall from the introduction that the λC-calculus is a variation of λP in which patterns are taken to be algebraic terms, and where multiple pattern matching is allowed by means of generalized abstractions of the form $\lambda P_1.M_1|\ldots|\lambda P_k.M_k$. The reduction relation is a generalization of that of λP:

$$(\lambda P_1.M_1|\ldots|\lambda P_k.M_k)P_i^\sigma \rightarrow M_i^\sigma$$

Confluence has been proved for linear constructor patterns, as well as for *rigid* multiple-patterns (i.e. patterns satisfying the *Rigid Pattern Condition* introduced in [30] for λP, and where the different P_i in $\lambda P_1.M_1|\ldots|\lambda P_k.M_k$ do not unify with each other). Since λC does not allow abstractions to be used as patterns, we may restrict our set of patterns to only variables and constructors applied to 0 or more arguments, which must themselves be patterns. In other words, the syntax for the patterns would be as follows:

$$P, P_1 \cdots P_n ::= x \mid \ CP_1 \cdots P_n$$

with $n \geq 0$. Note that this set of patterns satisfies RPC^{++}.

The syntax of CL_P is extended with *case* combinators in order to allow matching over multiple patterns, and even provide different responses depending on the pattern that has been matched. The case combinators have the form $\theta_{P_1,\ldots,P_n}$ with $n \geq 1$, where P_1,...,P_n are patterns which are not pairwise unifiable. A new rule schema is introduced:

$$\theta_{P_1,\ldots,P_n} x_1 \ldots x_n P_i \to x_i P_i, \quad \text{if } 1 \leq i \leq n$$

Definition 7.12 *The translation from λC to $CL_P + \theta$ (that is, CL_P extended with the* case *combinator and constructors) is as follows:*

$$\begin{array}{rcl}
x_{CL} & \triangleq & x \\
C_{CL} & \triangleq & C \\
(MN)_{CL} & \triangleq & M_{CL} N_{CL} \\
(\lambda P.N)_{CL} & \triangleq & \lambda^* P_{CL}.N_{CL} \\
(\lambda P_1.M_1 | \ldots | \lambda P_n.M_n)_{CL} & \triangleq & \theta_{(P_1)_{CL},\ldots,(P_n)_{CL}} (\lambda^*(P_1)_{CL}.(M_1)_{CL}) \ldots (\lambda^*(P_n)_{CL}.(M_n)_{CL}), \\
 & & \textit{if } n \geq 2
\end{array}$$

With λ^* defined as before (Definition 4.1).

Here is an example of how the translation works.

$$(\lambda <> . <> | \lambda \mathtt{cons}\, x\, y.y)(\mathtt{cons}\ x_1(\mathtt{cons}\ x_2 <>))$$

would translate to:

$$\theta_{<>,\mathtt{cons}\, x\, y}(K <>)(S_{\mathtt{cons}\, x\, y}(K(SKK))\Pi^2_{\mathtt{cons}\, x\, y})(\mathtt{cons}\ x_1(\mathtt{cons}\ x_2 <>))$$

which reduces as follows:

$$\begin{array}{ll}
\theta_{<>,\mathtt{cons}\, x\, y}(K <>)(S_{\mathtt{cons}\, x\, y}(K(SKK))\Pi^2_{\mathtt{cons}\, x\, y})(\mathtt{cons}\ x_1(\mathtt{cons}\ x_2 <>)) & \to_{W_P} \\
S_{\mathtt{cons}\, x\, y}(K(SKK))\Pi^2_{\mathtt{cons}\, x\, y}(\mathtt{cons}\ x_1(\mathtt{cons}\ x_2 <>)) & \to_{W_P} \\
K(SKK)(\mathtt{cons}\ x_1(\mathtt{cons}\ x_2 <>))(\Pi^2_{\mathtt{cons}\, x\, y}(\mathtt{cons}\ x_1(\mathtt{cons}\ x_2 <>))) & \to_{W_P} \\
SKK(\Pi^2_{\mathtt{cons}\, x\, y}(\mathtt{cons}\ x_1(\mathtt{cons}\ x_2 <>))) & \to_{W_P} \\
SKK(\mathtt{cons}\ x_2 <>) & \twoheadrightarrow_{W_P} \\
\mathtt{cons}\ x_2 <> . &
\end{array}$$

The requirement for the patterns in the case not to be pairwise unifiable is needed – as in λC – to ensure that the argument will match at most one pattern.

The argument which will be matched against the patterns is placed immediately after the case combinator to facilitate potential implementations, as well as translations to other combinatory calculi which handle the matching in a sequential manner. Since the number of required arguments depends on the number of patterns used within the case combinator, skipping through a varying (and arbitrarily high) number of arguments in order to find the term that will be matched against each pattern would be both difficult and inefficient.

We now prove simulation for the general case (Corollary. 7.15).

Lemma 7.13 $(M^\sigma)_{CL} = (M_{CL})^{\sigma_{CL}}$.

Proof.- First note that Lemma 4.8 still holds, since none of the new cases for the translations introduce nor erase free variables. We prove this result by induction on the size of M, which is now a λC-term.

For the cases where M is a variable, an application or a single pattern abstraction, the proof is analogous to that of Lemma 4.9. If M is a constructor, the result holds trivially, since $(M^\sigma)_{CL} = M_{CL} = M = (M_{CL})^{\sigma_{CL}}$.

If $M = \lambda P_1.M_1| \ldots |\lambda P_k.M_k$, then $M^\sigma = (\lambda P_1.M_1)^\sigma| \ldots |(\lambda P_n.M_n)^\sigma$ and, by IH, $((\lambda P_i.M_i)^\sigma)_{CL} = ((\lambda P_i.M_i)_{CL})^{\sigma_{CL}}$ for $i \in \{1, \ldots, n\}$.

$$\begin{array}{lcl} (M^\sigma)_{CL} & = & \\ \theta_{(P_1)_{CL},\ldots,(P_n)_{CL}}((\lambda P_1.M_1)_{CL})^{\sigma_{CL}} \ldots ((\lambda P_n.M_n)_{CL})^{\sigma_{CL}} & = & \\ \theta_{(P_1)_{CL},\ldots,(P_n)_{CL}}(\lambda^*(P_1)_{CL}.(M_1)_{CL})^{\sigma_{CL}} \ldots (\lambda^*(P_n)_{CL}.(M_n)_{CL})^{\sigma_{CL}} & = & (M_{CL})^{\sigma_{CL}}. \end{array}$$

Keep in mind that substitutions in λC, as in λP, do not affect the pattern of an abstraction, since its variables are bound. This means that it is safe to use $(P_1)_{CL}, \ldots, (P_n)_{CL}$ as subindices for θ without applying the substitution to these patterns. ■

LEMMA 7.14 *$\theta_{P_1,\ldots,P_n}(\lambda^* P_1.M_1) \ldots (\lambda^* P_n.M_n)P_i^\sigma \twoheadrightarrow_{W_P} M_i^\sigma$ for $1 \leq i \leq n$, if $\mathsf{dom}(\sigma) \subseteq \mathsf{FV}(P_i)$.*

Proof.- First note that Lemma 4.14 still holds, since the definition of λ^* has not changed.

$\theta_{P_1,\ldots,P_n}(\lambda^* P_1.M_1) \ldots (\lambda^* P_n.M_n)P_i^\sigma \to_{W_P} (\lambda^* P_n.M_n)P_i^\sigma$ by the new reduction rule. By Lemma 4.14, $(\lambda^* P_n.M_n)P_i^\sigma \twoheadrightarrow_{W_P} M_i^\sigma$. ■

COROLLARY 7.15 *$((\lambda P_1.M_1| \ldots |\lambda P_k.M_k)P_i^\sigma)_{CL} \twoheadrightarrow_{W_P} M_i^\sigma$ if $\mathsf{dom}(\sigma) \subseteq \mathsf{FV}(P_i)$.*

Proof.- If $n \geq 2$, then by Lemma 7.13, $(P_i^\sigma)_{CL} = ((P_i)_{CL})^{\sigma_{CL}}$.

$$\begin{array}{lcl} ((\lambda P_1.M_1| \ldots |\lambda P_k.M_k)P_i^\sigma)_{CL} & = & \\ (\lambda P_1.M_1| \ldots |\lambda P_k.M_k)_{CL}((P_i)_{CL})^{\sigma_{CL}} & = & \\ \theta_{(P_1)_{CL},\ldots,(P_n)_{CL}}(\lambda^*(P_1)_{CL}.(M_1)_{CL}) \ldots (\lambda^*(P_n)_{CL}.(M_n)_{CL})((P_i)_{CL})^{\sigma_{CL}}. & & \end{array}$$

The result follows by Lemma 7.14.

If $n = 1$, the result holds by Lemmas 7.13 and 4.14, as in Corollary 4.15. ■

Generalized pattern matching can also be used to introduce structural polymorphism into our calculus. For example, we can now define a term that returns the first argument of a data structure defined by the application of a constructor to two terms (a list with a first element followed by another list, a tree made by combining two trees, etc.), and returns the data structure unchanged if it has fewer than two arguments (and becomes blocked if the argument has more than two arguments or is not a data structure). If we introduce a new pattern ♠ and extend the matching predicate so that ♠ is matched by

every constructor (and only by constructors), then the result will be achieved by the term $\theta_{\spadesuit xy,\spadesuit z,\spadesuit}(S(K\Pi^2_{\spadesuit w})\Pi^1_{\spadesuit uv})II$. This technique can be applied with other structures and patterns, achieving an effect resembling path polymorphism (the difference is that we're using an ad-hoc matching predicate that is external to the calculus in order to produce this effect). For example, we can now define a term that returns the first argument of a data structure defined by the application of a constructor to two terms (a list with a first element followed by another list, a tree made by combining two trees, etc.), and returns the data structure unchanged if it has fewer than two arguments (and becomes blocked if the argument has more than two arguments or is not a data structure).

In order to define a pattern that is matched by data structures in general, the matching predicate can be extended so that every term of the form $CP_1...P_n$, with C a constructor and $n \geq 0$, matches the new pattern. Since terms of this form may only have internal reductions, this extension is coherent and thus confluence still holds. Data structures which are not of the form $CP_1...P_n$ do not present a problem, as they can be reduced to that form and then matched against the pattern.

The SF-calculus. Finally, there is another variant which can be used to capture other forms of polymorphism and partially simulate the SF-calculus. It consists of introducing two new special patterns (@ and $\circ$) in addition to the case combinator θ. However, we only need to use the θ combinator with these two patterns, so we can simply introduce one form of the case combinator, namely $\theta_{\circ,@}$. In this extension we also allow the use of the combinators $\Pi^1_@$ and $\Pi^2_@$, which will allow us to decompose a wide range of applications.

We extend the original matching algorithm so that the pattern @ is matched by all (instances of) application patterns. That is, every term of the forms K_PM, S_PM, $S_PM_1M_2$ and, eventually, all applied constructors. This way the @ pattern will be capable of dealing with applications, in spite of the presence of infinitely many constants. We are assuming the use of matching-safe patterns.

The pattern $\circ$ will be matched by all constants, namely K_P, S_P, Π^1_{PQ}, Π^2_{PQ}, $\Pi^1_@$, $\Pi^2_@$ and eventually all constructors without their arguments. Once more, a single pattern can 'absorb' the roles of infinitely many patterns.

Given this extension, we define:

$$\hat{\mathrm{F}} \triangleq S(KK)(S(S(KS)(S(K(S(SKK)))(S(KK)\Pi^1_@)))(S(KK)\Pi^2_@))$$

This term, also a pattern, is the result of unfolding the expression

$$\lambda^* x.\lambda^* y.\lambda^* z.z(\Pi^1_@ x)(\Pi^2_@ x).$$

When applied to an instance of an application pattern $(PQ)^\sigma$, it reduces in many steps to $K(S(SI(KP^\sigma))(KQ^\sigma))$. This way, a term of the form $\hat{\mathrm{F}}(PQ)^\sigma MN$ will reduce in many steps to $NP^\sigma Q^\sigma$.

This becomes useful when we use $\hat{\mathrm{F}}$ as an argument for the case combinator. The term $\theta_{\circ,@}(KK)\hat{\mathrm{F}}$ can partially simulate the behavior of the combinator F from the SF-calculus, if we consider all instances of application patterns as "*factorable forms*". The reduction rules of the SF-calculus are the following:

$$\begin{array}{lcll} SMNX & \to & MX(NX) & \\ FOMN & \to & M & \text{if } O \text{ is a constant.} \\ F(PQ)MN & \to & NPQ & \text{if } PQ \text{ is a factorable form.} \end{array}$$

REMARK 7.16 *If we define F as $\theta_{\circ,@}(KK)\hat{F}$, we can simulate those same reductions in many steps. It is not, however, a full simulation of the SF-calculus, for the following reasons: first, while the SF-calculus has only 2 constants, we have an infinite number of them. Second, the term F is a constant in the SF-calculus, while in our extension it is a factorable form. This means that reducing the term $FFMN$ will yield different results in each calculus, and thus ours is not an exact simulation. It would be possible to patch the matching algorithm in order to make $\hat{F}$ match the pattern $\circ$ and not @, but such a modification is not necessary to achieve the distinction between atoms and compounds: the term $\texttt{isComp} = \lambda x.Fx(KI)(K(KK))$ defined in [21] can also be defined here as $\lambda^* x.Fx(KI)(K(KK))$ for our definition of F. Decomposition of "factorable forms" is already achieved in our system by projectors, thus we can already have a measure of structural polymorphism.*

This extension is coherent for the same reason as the ♠ extension (terms which match the pattern @ can only have internal reductions, while terms that match the pattern $\circ$ cannot be reduced), thus confluence still holds (note that the rules for the θ combinators do not overlap with each other – since matching against @ and $\circ$ are mutually exclusive –, nor with any of the original rules).

8 Comparison between CL_P and other Pattern Calculi

There are several other calculi which can handle patterns in different ways. We will now briefly compare CL_P with some of these calculi.

The Pure Pattern Calculus ([19, 23]) is more complex in its formulation, requiring a concept of matchable forms with their own specific syntax. The matching mechanism is also handled by an external algorithm, defined as a sequence of clauses with complex conditions and a specific order of evaluations. The concept of variable is also less intuitive, as each variable x has a counterpart $\hat{x}$ and an associated constructor.

The logical frameworks defined in [17] rely on external algorithms in order to determine the matching process. It generalizes the settings of some of the previous works.

It contemplates the possibility of detecting whether the term to be reduced is open or ground.

While the need to match a given term against a pattern is made explicit in both Rho-calculus ([9]) and the ([32]), matching itself is resolved by an external algorithm; while in CL_P the matching is handled by the TRS itself. Generalizations or variations to the matching process may be introduced by either adding or modifying the reduction rules, and the system will remain confluent as long as orthogonality is preserved. [32] also proves strong normalization of typable terms.

In all the above cases the matching, once formulated, is instantaneous. And the concept of bound variable is needed in all of the previous rewriting systems mentioned above.

The SF-calculus, mentioned in the introduction, is a combinatory calculus which can look into arguments much like in all known pattern calculi. However, instead of using patterns, it defines a concept of factorable form, which can be decomposed by a conditional combinator F. This combinator has the ability to distinguish between atoms and compound terms, allowing it to define a number of functions based on the structure of the arguments. On the other hand, CL_P is a TRS, with atomic combinators and no conditional rules, and the presence of projectors makes the decomposition of arguments in our calculus more intuitive. The SF-calculus could be mapped into a TRS by replacing the F rules by an infinite number of rules (one for each constant and one for each application pattern which can be matched by factorable forms, the latter being an infinite set). Additionally, [21] does not present a mapping between the SF-calculus and a higher-order pattern calculus; nor does it present a simple type system for the calculus, since simple types have been shown to have insufficient expressive power to type the F and E combinators. Instead, a system with polymorphic types is introduced, with the drawback of type assignment being undecidable. [7] introduces a type system with a decidable type-assignment for a reduced version of the calculus named the $SFBY$-calculus, which emulates the E combinator via a term constructed with several hundred operators.

9 Further Work and Conclusion

In this article we have presented a system of combinatory logic (CL_P) for a λ-calculus (λP) in which functions may be abstracted over a general notion of pattern that includes applications and abstractions themselves. We have proved that the associated notion of pattern-matching can be handled within the context of a Term Rewriting System (TRS), that is, by means of first-order rewriting. For that, two issues have been addressed:

1. Emulating successful matching in λP by means of combinators; and

2. Computing the resulting substitution, also by means of combinators which have to decompose applicative terms to achieve the desired result.

We have also worked out the necessary restrictions in order to avoid ill-formed patterns which may break confluence. This has been achieved by introducing syntactic characterizations of such restrictions. Moreover, given a term or pattern, these restrictions can be efficiently verified. We have also presented a type system based on simple types, for which we proved normalization of typable terms. Finally, a number of extensions have been addressed of which $CL_\star$ stands out. In contrast to CL_P, which although presented in terms of a finite number of rule schemas has an infinite number of rule instances, $CL_\star$ is a finite TRS. This is possible by encoding matching within the calculus itself, albeit at the cost of complicating the combinator syntax.

We mention some possible avenues to explore.

Combinators for dynamic patterns. The Pure Pattern Calculus or PPC [19, 22, 23] is a pattern calculus in which patterns may be created during runtime. For example, consider the PPC-term M:

$$x \hookrightarrow_x (x \hookrightarrow \texttt{true} | y \hookrightarrow \texttt{false})$$

The symbol "$\hookrightarrow$" allows abstractions to be built, its first argument being the pattern and the second one the body. A set of variable subscripts, such as x in "$\hookrightarrow_x$", indicates which variables in the pattern are considered matching variables (x in this example) and which are considered free variables (these are to be replaced from the "outside"). The order among the patterns is important, as reduction will proceed according to the first match. The PPC-term $M(\texttt{Succ 0})(\texttt{Succ 0})$ reduces in two steps to `true`. Note that, after the first step, (`Succ 0`) becomes a new pattern producing:

$$\texttt{Succ 0} \hookrightarrow \texttt{true} | y \hookrightarrow \texttt{false}$$

Similarly, $M(\texttt{Succ 0})\texttt{0}$ reduces to `false`. It would be interesting to obtain a combinatory account of PPC.

Higher-order λP-unification via combinators. We are interested in studying unification with respect to CL_P-equality [14]. The formulation of a higher-order unification algorithm turns out to be complicated (even when restricting to simply typed λ-calculus terms), due to the presence of bound variables. For this reason, [14] emphasizes the importance of considering the conversion to combinatory logic as an intermediate step, thus eliminating bound variables. In this way the formulation of the algorithm is simplified considerably, according to [14]. Note that the absence of bound variables is the only requirement for a calculus to serve this purpose. Our system of combinators seems to be an adequate language due to the fact that all required properties extend the classical ones.

Detecting matching failure through type checking. It would be interesting to study other possible type systems of CL_P, particularly dependent type systems in which the patterns play a role in detecting matching failure. The rules would have to be chosen carefully to account for undecided matching. For instance, they should allow the correct typing of $S_P M N P^\sigma$, taking into account that the normal form of NP^σ cannot be known in advance.

Acknowledgements: We are grateful to Eduardo Bonelli for his help and support.

References

[1] Ariel Arbiser. *Explicit substitution systems and subsystems*. PhD thesis, Universidad de Buenos Aires, 2005.

[2] Ariel Arbiser, Alexandre Miquel, and Alejandro Ríos. A lambda-calculus with constructors. In Frank Pfenning, editor, *Term Rewriting and Applications*, volume 4098 of *Lecture Notes in Computer Science*, pages 181–196. Springer Berlin Heidelberg, 2006.

[3] Ariel Arbiser and Gabriela Steren. Combinatory logics for lambda calculi with patterns. 2010.

[4] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA, 1998.

[5] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Sole Distributors for the U.S.A. And Canada, Elsevier Science Pub. Co., 1984.

[6] Henk P. Barendregt. Handbook of logic in computer science (vol. 2). chapter Lambda Calculi with Types, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992.

[7] Alexander Bates and Reuben N.S. Rowe. Structural types for the factorisation calculus. Master's thesis, University College London, 2014.

[8] Mark Bezem, Jan Willem Klop, and Roel de Vrijer. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.

[9] Horatiu Cirstea and Claude Kirchner. The rewriting calculus — Part I. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.

[10] Haskell Brooks Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934.

[11] Haskell Brooks Curry. Modified basic functionality in combinatory logic. *Dialectica*, 23(2):83–92, 1969.

[12] Haskell Brooks Curry and Robert Feys. *Combinatory Logic*. Number v. 1 in Combinatory Logic. North-Holland Publishing Company, 1958.

[13] Haskell Brooks Curry, Jonathan P. Seldin, and J. Roger Hindley. *Combinatory Logic, 2*. Number v. 2 in Studies in logic and the foundations of mathematics. North-Holland, 1972.

[14] Daniel J. Dougherty. Higher-order unification via combinators. *Theoretical Computer Science*, 114:273–298, 1993.

[15] Marcelo Finger. Towards structurally-free theorem proving. *Logic Journal of IGPL*, 6(3):425–449, 1998.

[16] Dov M. Gabbay and et al. Extending the curry-howard interpretation to linear, relevance and other resource logics. *JOURNAL OF SYMBOLIC LOGIC*, 1992.

[17] Furio Honsell. A framework for defining logical frameworks. In *University of Udine*, 2006.

[18] C Barry Jay. Distinguishing data structures and functions: the constructor calculus and functorial types. In *Typed Lambda Calculi and Applications*, pages 217–239. Springer, 2001.

[19] C. Barry Jay. Typing first-class patterns. In *In Higher-Order Rewriting, proceedings*, 2006.

[20] C. Barry Jay. *Pattern Calculus: Computing with Functions and Structures*. Springer, 2009.

[21] C. Barry Jay and Thomas Given-Wilson. A combinatory account of internal structure. *Journal of Symbolic Logic*, 76(3):807–826, 09 2011.

[22] C. Barry Jay and Delia Kesner. Pure pattern calculus. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 262–274, 2005.

[23] C. Barry Jay and Delia Kesner. First-class patterns. *J. Funct. Program.*, 19(2):191–225, March 2009.

[24] Jan Willem Klop. A counterexample to the cr property for λ-calculus+ dmm m. *Typed note, Utrecht University*, 1976.

[25] Jan Willem Klop, Vincent van Oostrom, and Roel de Vrijer. Lambda calculus with patterns. *Theoretical Computer Science*, 398(13):16 – 31, 2008. Calculi, Types and Applications: Essays in honour of M. Coppo, M. Dezani-Ciancaglini and S. Ronchi Della Rocca.

[26] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[27] Moses Ilyich Schönfinkel. Über die bausteine der mathematischen logik. *Mathematische Annalen*, 92(3-4):305–316, 1924.

[28] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science, 2006.

[29] William W. Tait. Intensional interpretations of functionals of finite type i. *Journal of Symbolic Logic*, 32(2):198–212, 06 1967.

[30] Vincent van Oostrom. Lambda calculus with patterns. Technical report, Vrije Universiteit, 1990.

[31] Benjamin Wack. *Typage et déduction dans le calcul de réécriture*. PhD thesis, Université Henri Poincaré-Nancy I, 2005.

[32] Benjamin Wack and Clement Houtmann. Strong Normalization in two Pure Pattern Type Systems. *Mathematical Structures in Computer Science*, 18(3):431–465, 2008.

Ariel Arbiser
Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires, Argentina
E-mail: arbiser@dc.uba.ar

Gabriela Steren
Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires, Argentina
E-mail: gsteren@yahoo.com