Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 351 (2020) 95–113

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Pure Pattern Calculus *à la* de Bruijn[1]

Alexis Martín

Universidad de Buenos Aires, Argentina

Alejandro Ríos

Universidad de Buenos Aires, Argentina

Andrés Viso

Universidad de Buenos Aires, Argentina
Universidad Nacional de Quilmes, Argentina

Abstract

It is well-known in the field of programming languages that dealing with variable names and binders may lead to conflicts such as undesired captures when implementing interpreters or compilers. This situation has been overcome by resorting to de Bruijn indices for calculi where binders capture only one variable name, like the λ-calculus. The advantage of this approach relies on the fact that so-called α-equivalence becomes syntactical equality when working with indices.
In recent years pattern calculi have gained considerable attention given their expressiveness. They turn out to be notoriously convenient to study the foundations of modern functional programming languages modeling features like pattern matching, path polymorphism, pattern polymorphism, etc. However, the literature falls short when it comes to dealing with α-conversion and binders capturing simultaneously several variable names. Such is the case of the *Pure Pattern Calculus* (PPC): a natural extension of λ-calculus that allows to abstract virtually any term.
This paper extends de Bruijn's ideas to properly overcome the multi-binding problem by introducing a novel presentation of PPC with bidimensional indices, in an effort to implement a prototype for a typed functional programming language based on PPC that captures path polymorphism.

Keywords: de Bruijn indices, pattern calculi, pattern matching, α-equivalence.

1 Introduction

The foundations of functional programming languages like LISP, Miranda, Haskell or the ones in the ML family (Caml, SML, OCaml, etc.) rely strongly on the study of the λ-calculus [2] and its many variants introduced over the years. Among them there are the *pattern calculi* [23,7,16,6,14,12,18], whose key feature can be identified as *pattern-matching*. Pattern-matching has been extensively used in programming

[1] This work was partially supported by LIA INFINIS, and the ECOS-Sud program PA17C01.

https://doi.org/10.1016/j.entcs.2020.08.006

languages as a means for writing succinct and elegant programs. It stands for the possibility of defining functions by cases, analysing the shape of their arguments, while providing a syntactic tool to decompose such arguments in their parts when applying the function.

In the standard λ-calculus, functions are represented by expressions of the form $\lambda x.t$, where x is the formal parameter and t the body of the function. Such a function may be applied to any term, regardless of its form, as dictated by the β-reduction rule: $(\lambda x.t)\, u \mapsto_\beta \{x \setminus u\}t$, where $\{x \setminus u\}t$ stands for the result of replacing all free occurrences of x in t by u. Note that no requirement on the shape of u is placed. Pattern calculi, on the contrary, provide generalisations of the β-reduction rule in which abstractions $\lambda x.t$ are replaced by more general terms like $\lambda p.t$ where p is called a *pattern*. For example, consider the function $\lambda\langle x, y\rangle.x$ that projects the first component of a pair. Here the pattern is the pair $\langle x, y\rangle$ and the expression $(\lambda\langle x, y\rangle.x)\, u$ will only be able to reduce if u is indeed of the form $\langle u_1, u_2\rangle$. Otherwise, reduction will be blocked.

We are particularly interested in studying the *Pure Pattern Calculus* (PPC) [15] and the novel features it introduced in the field of pattern calculi, namely *path polymorphism* and *pattern polymorphism*. The former refers to the possibility of defining functions that uniformly traverse arbitrary data structures, while the latter allows to consider patterns as parameters that may be dynamically generated in run-time. Developing such a calculus implies numerous technical challenges to guarantee well-behaved operational semantics in the untyped framework. Recently, a static type system has been introduced for a restriction of PPC called *Calculus of Applicative Patterns* (CAP) [28], which is able to capture the path polymorphic aspect of PPC. Moreover, type-checking algorithms for such a formalism has also been studied [10], as a first step towards an implementation of a prototype for a typed functional programming language capturing such features. Following this line of research, studies on the definition of normalising strategies for PPC have been done as well [3,4]. Such results are ported to CAP by means of a simple embedding [27] where the static typing discipline gives further guarantees on the well-behaved semantics of terms.

Within this framework, the present work aims to throw some light on the implementation aspects of these formalisms. In particular, working modulo α-conversion [2] implies dealing with variable renaming during the implementation. Such an approach is known to be error-prone and computationally expensive. One way of getting rid of this problem in the λ-calculus setting is adopting de Bruijn notation [8,9], a technique that simply avoids working modulo α-conversion. To the best of our knowledge, no dynamic pattern calculi in the likes of PPC with de Bruijn indices has been formalised in the literature. However, there are some references worth mentioning. In [24] an alternative presentation of PPC is given in the framework of *Higher-Order Pattern Rewriting System* (HRS) [22,20], together with translations between the two systems. On the other hand, in [5] de Bruijn ideas had been extended to *Expression Reduction Systems* (ERS) [11] also providing formal translations from systems with names to systems with indices, and vice-versa. Moreover, the correspondence between HRS and ERS has already been established [25]. The composition of such translations might derive a higher order system *à la* de Bruijn capturing the features of PPC. However, this would result in a rather indirect solution to our problem where many technicalities still need to be sorted out.

We aim to formalise an intuitive variant of PPC with de Bruijn indices where known results for the original calculus, such as the existence of normalising strategies, may easily be ported and reused.

1.1 Contributions

This paper extends de Bruijn's ideas to handle binders that capture multiple symbols at once, by means of what we call *bidimensional indices*. These ideas are illustrated by introducing a novel presentation of PPC, without variable/matchable names, called $\mathsf{PPC}_{\mathsf{dB}}$. Moreover, binders in the new proposed calculus are capable of handling two kinds of indices, namely variable and matchable indices, as required by the PPC operational semantics.

Proper translations from PPC to $\mathsf{PPC}_{\mathsf{dB}}$ and back are introduced. This functions preserve the matching operation and, hence, the operational semantics of both calculi. Moreover, they turn out to be the inverse of each other. This leads to a crucial strong bisimulation result between the two calculi, which allows to import many known properties of PPC into $\mathsf{PPC}_{\mathsf{dB}}$, for instance confluence and the existence of normalising strategies.

1.2 Structure of the paper

We start by briefly introducing PPC and reminding the mechanism of de Bruijn indices for the λ-calculus in Sec. 2. The novel $\mathsf{PPC}_{\mathsf{dB}}$ is formalised in Sec. 3, followed by the introduction of the translations in Sec. 4. The strong bisimulation result is presented in Sec. 5 together with a discussion of different properties of $\mathsf{PPC}_{\mathsf{dB}}$ that follow from it. We conclude in Sec. 6 and discuss possible lines of future work. Some technical details and proofs are relegated to the extended report available online [19].

2 Preliminaries

This section introduces preliminary concepts that guide our development and will help the reader follow the new ideas presented in this work.

2.1 The Pure Pattern Calculus

We start by briefly introducing the *Pure Pattern Calculus* (PPC) [15], an extension of the λ-calculus where virtually any term can be abstracted. This gives place to two versatile forms of polymorphism that set the foundations for adding novel features to future functional programming languages: namely *path polymorphism* and *pattern polymorphism*. This work, however, focuses on implementation related aspects of PPC and will not delve deeper into these new forms of polymorphism. We refer the reader to [15,13] for an in-depth study of them.

Given an infinitely countable set of *symbols* $\mathbb{V}$ $(x, y, z, \ldots)$, the sets of *terms* $\mathbb{T}_{\mathsf{PPC}}$ and *contexts* are given by the following grammar:

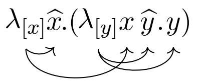
$$\textbf{Terms } t ::= x \mid \widehat{x} \mid t\,t \mid \lambda_\theta t.t \qquad \textbf{Contexts } \mathtt{C} ::= \Box \mid \mathtt{C}\,t \mid t\,\mathtt{C} \mid \lambda_\theta \mathtt{C}.t \mid \lambda_\theta t.\mathtt{C}$$

where θ is a list of symbols that are bound by the abstraction. A symbol x appearing in a term is dubbed a *variable symbol* while $\widehat{x}$ is called a *matchable symbol.* In particular, given $\lambda_\theta p.t$, θ binds variable symbols in the *body* t and matchable symbols in the *pattern* p. Thus, the set of *free variables* and *free matchables* of a term t, written $\mathsf{fv}(t)$ and $\mathsf{fm}(t)$ respectively, are inductively defined as:

$$\begin{aligned} \mathsf{fv}(x) &\triangleq \{x\} & \mathsf{fm}(x) &\triangleq \emptyset \\ \mathsf{fv}(\widehat{x}) &\triangleq \emptyset & \mathsf{fm}(\widehat{x}) &\triangleq \{x\} \\ \mathsf{fv}(t\,u) &\triangleq \mathsf{fv}(t) \cup \mathsf{fv}(u) & \mathsf{fm}(t\,u) &\triangleq \mathsf{fm}(t) \cup \mathsf{fm}(u) \\ \mathsf{fv}(\lambda_\theta p.t) &\triangleq \mathsf{fv}(p) \cup (\mathsf{fv}(t) \setminus \theta) & \mathsf{fm}(\lambda_\theta p.t) &\triangleq (\mathsf{fm}(p) \setminus \theta) \cup \mathsf{fm}(t) \end{aligned}$$

A term is said to be *closed* if it has no free variables. Note that free matchables are allowed, and should be understood as *constants* or *constructors* for data structures. The pattern p of an abstraction $\lambda_\theta p.t$ is *linear* if every symbol $x \in \theta$ occurs at most once in p.

To illustrate how variables and matchables are bound, consider the function elim defined as $\lambda_{[x]}\widehat{x}.(\lambda_{[y]} x\,\widehat{y}.y)$. The inner abstraction binds the only occurrence of the matchable $\widehat{y}$ in the pattern $x\,\widehat{y}$ and that of the variable y in the body y. However, the occurrence of x in $x\,\widehat{y}$ is not bound by the inner abstraction, as it is excluded from $[y]$, acting as a place-holder in that pattern. It is the outermost abstraction that binds both x in the inner pattern and $\widehat{x}$ in the outermost pattern. This is graphically depicted above.



A *substitution* $(\sigma, \rho, \ldots)$ is a partial function from variables to terms. The substitution $\sigma = \{x_i \setminus u_i\}_{i \in I}$, where I is a set of indices, maps the variable x_i into the term u_i (*i.e.* $\sigma(x_i) \triangleq u_i$) for each $i \in I$. Thus, its *domain* and *image* are defined as $\mathsf{dom}(\sigma) \triangleq \{x_i\}_{i \in I}$ and $\mathsf{img}(\sigma) \triangleq \{u_i\}_{i \in I}$ respectively. For convenience, a substitution σ is usually turned into a total function by defining $\sigma(x) \triangleq x$ for every $x \notin \mathsf{dom}(\sigma)$. Then, the identity substitution is denoted $\{\}$ or simply id.

A *match* $(\mu, \nu, \ldots)$ may be successful (yielding a substitution), it may fail (returning a special symbol `fail`) or be undetermined (denoted by a special symbol `wait`). The cases of success and failure are called *decided matches.* All concepts and notation relative to substitutions are extended to matches so that, for example, the domain of `fail` is empty while that of `wait` is undefined. The sets of free variable and free matchable symbols of σ are defined as the union of $\mathsf{fv}(\sigma x)$

and $\mathsf{fm}(\sigma x)$ for every $x \in \mathsf{dom}(\sigma)$ respectively, while $\mathsf{fv}(\mathtt{fail}) = \mathsf{fm}(\mathtt{fail}) = \emptyset$ and they are undefined for $\mathtt{wait}$. The set of symbols of a substitution is defined as $\mathsf{sym}(\sigma) \triangleq \mathsf{dom}(\sigma) \cup \mathsf{fv}(\sigma) \cup \mathsf{fm}(\sigma)$. The predicate x $\mathtt{avoids}$ σ states that $x \notin \mathsf{sym}(\sigma)$. It is extended to sets and matches as expected. In particular, θ $\mathtt{avoids}$ μ implies that μ must be decided.

The result of applying a substitution σ to a term t, denoted σt, is inductively defined as:

$$\sigma x \triangleq \sigma(x) \qquad\qquad \sigma(t\,u) \triangleq \sigma t\, \sigma u$$

$$\sigma \widehat{x} \triangleq \widehat{x} \qquad\qquad \sigma \lambda_\theta p.t \triangleq \lambda_\theta \sigma p.\sigma t \quad \text{if } \theta \texttt{ avoids } \sigma$$

The restriction in the case of the abstraction is required to avoid undesired captures of variables/matchables. However, it can always be satisfied by resorting to α-conversion.

The result of applying a match μ to a term t, denoted $\mu\, t$, is defined as: (i) if $\mu = \sigma$ a substitution, then $\mu\, t \triangleq \sigma t$; (ii) if $\mu = \mathtt{fail}$, then $\mu\, t \triangleq \lambda_{[x]}\widehat{x}.x$ (*i.e.* the identity function); or (iii) if $\mu = \mathtt{wait}$, then $\mu\, t$ is undefined.

The *composition* $\sigma \circ \sigma'$ of substitutions is defined as usual, *i.e.* $(\sigma \circ \sigma')x \triangleq \sigma(\sigma' x)$, and the notion is extended to matches by defining $\mu \circ \mu' \triangleq \mathtt{fail}$ if any of the two matches is $\mathtt{fail}$. Otherwise, if at least one of the two is $\mathtt{wait}$, then $\mu \circ \mu' \triangleq \mathtt{wait}$. In particular, $\mathtt{fail} \circ \mathtt{wait} = \mathtt{fail}$. The *disjoint union* $\mu \uplus \mu'$ of matches is defined as follows: (i) if $\mu = \mathtt{fail}$ or $\mu' = \mathtt{fail}$, then $\mu \uplus \mu' \triangleq \mathtt{fail}$; else (ii) if $\mu = \mathtt{wait}$ or $\mu' = \mathtt{wait}$, then $\mu \uplus \mu' \triangleq \mathtt{wait}$; otherwise (iii) both μ and μ' are substitutions and if $\mathsf{dom}(\mu) \cap \mathsf{dom}(\mu') \neq \emptyset$, then $\mu \uplus \mu' \triangleq \mathtt{fail}$, else:

$$(\mu \uplus \mu')x \triangleq \begin{cases} \mu x & \text{if } x \in \mathsf{dom}(\mu) \\ \mu' x & \text{if } x \in \mathsf{dom}(\mu') \\ x & \text{otherwise} \end{cases}$$

Disjoint union is used to guarantee that the matching operation is deterministic.

Before introducing the matching operation it is necessary to motivate the concept of *matchable form*. The pattern $\widehat{x}\,\widehat{y}$ allows, at first, to decompose arbitrary applications, which may lead to the loss of confluence. For instance:

$$(\lambda_{[x,y]}\widehat{x}\,\widehat{y}.y)\,((\lambda_{[w]}\widehat{w}.\widehat{z_0}\,\widehat{z_1})\,\widehat{z_0}) \to (\lambda_{[x,y]}\widehat{x}\,\widehat{y}.y)\,(\widehat{z_0}\,\widehat{z_1}) \to \widehat{z_1}$$

$$(\lambda_{[x,y]}\widehat{x}\,\widehat{y}.y)\,((\lambda_{[w]}\widehat{w}.\widehat{z_0}\,\widehat{z_1})\,\widehat{z_0}) \to \widehat{z_0}$$

This issue arises when allowing to match the pattern $\widehat{x}\,\widehat{y}$ with an application that may still be reduced, like the argument $(\lambda_{[w]}\widehat{w}.\widehat{z_0}\,\widehat{z_1})\,\widehat{z_0}$ of the outermost redex in the example above. To avoid this situation it is required for the match to be decided only if the argument is sufficiently evaluated. An analogous issue occurs if the pattern is reducible. Thus, both the pattern and the argument must be in matchable form for the match to be decided. The set of *data structures* $\mathbb{D}_{\mathsf{PPC}}$ and *matchable forms*

$\mathbb{M}_{\mathsf{PPC}}$ are given by the following grammar:

$$\textbf{Data structures } d ::= \widehat{x} \mid d\,t \qquad \textbf{Matchable forms } m ::= d \mid \lambda_\theta t.t$$

The *matching operation* $\{\!\{p \setminus^\theta u\}\!\}$ of a pattern p against a term u relative to a list of symbols θ is defined as the application, in order, of the following equations:

$$\begin{array}{rcll}
\{\!\{\widehat{x} \setminus^\theta u\}\!\} & \triangleq & \{x \setminus u\} & \text{if } x \in \theta \\
\{\!\{\widehat{x} \setminus^\theta \widehat{x}\}\!\} & \triangleq & \{\} & \text{if } x \notin \theta \\
\{\!\{p\,q \setminus^\theta t\,u\}\!\} & \triangleq & \{\!\{p \setminus^\theta t\}\!\} \uplus \{\!\{q \setminus^\theta u\}\!\} & \text{if } t\,u, p\,q \in \mathbb{M}_{\mathsf{PPC}} \\
\{\!\{p \setminus^\theta u\}\!\} & \triangleq & \mathtt{fail} & \text{if } u, p \in \mathbb{M}_{\mathsf{PPC}} \\
\{\!\{p \setminus^\theta u\}\!\} & \triangleq & \mathtt{wait} & \text{otherwise}
\end{array}$$

An additional check is imposed, namely $\mathsf{dom}(\{\!\{p \setminus^\theta u\}\!\}) = \theta$. Otherwise, $\{\!\{p \setminus^\theta u\}\!\} \triangleq$ `fail`. This last condition is necessary to prevent bound symbols from going out of scope when reducing. It can be easily guaranteed though by requesting, for each abstraction $\lambda_\theta p.t$, that $\theta \subseteq \mathsf{fm}(p)$. For instance, consider the term $(\lambda_{[x,y]}\widehat{x}.y)\,u$. Without this final check, matching the argument u against the pattern $\widehat{x}$ would yield a substitution $\{x \setminus u\}$ and no term would be assigned to the variable y in the body of the abstraction.

Finally, the reduction relation $\rightarrow_{\mathsf{PPC}}$ of PPC is given by the closure by contexts of the rewriting rule:

$$(\lambda_\theta p.s)\,u \mapsto_{\mathsf{PPC}} \{\!\{p \setminus^\theta u\}\!\}\,s$$

whenever $\{\!\{p \setminus^\theta u\}\!\}$ is a decided match. To illustrate the operational semantics of PPC consider the term elim introduced above, applied to the function $\lambda_{[z]}\widehat{z}.\widehat{c}\,z\,\widehat{n}$ where the free matchables $\widehat{c}$ and $\widehat{n}$ can be seen as constructors for lists cons and nil respectively:

$$(\lambda_{[x]}\widehat{x}.(\lambda_{[y]}x\,\widehat{y}.y))\,(\lambda_{[z]}\widehat{z}.\widehat{c}\,z\,\widehat{n}) \quad \rightarrow_{\mathsf{PPC}} \quad \lambda_{[y]}(\lambda_{[z]}\widehat{z}.\widehat{c}\,z\,\widehat{n})\,\widehat{y}.y \quad \rightarrow_{\mathsf{PPC}} \quad \lambda_{[y]}\widehat{c}\,\widehat{y}\,\widehat{n}.y$$

In the first step, $\lambda_{[z]}\widehat{z}.\widehat{c}\,z\,\widehat{n}$ is substituted for x into the pattern $x\,\widehat{y}$. In the second step, the resulting application, which resides in the pattern, is reduced. The resulting term, when applied to an argument, will yield a successful matching only if this argument is a compound data of the form $\widehat{c}\,t\,\widehat{n}$.

This relation is shown to be *confluent* (CR) based on the matching operation introduced above.

Theorem 2.1 **([15])** *The reduction relation $\rightarrow_{\mathsf{PPC}}$ is confluent (CR).*

2.2 de Bruijn indices

We introduce next de Bruijn indices for the λ-calculus. Among the many presentations of de Bruijn indices in the literature, we will follow that of [17] as our development builds upon their ideas. In particular, we choose to work with the presentation where indices are partially updated as the term is being traversed by the substitution operation (details below). We refer the reader to [17] for the equivalent version where the update is performed once at the end of the substitution process. We introduce now the *λ-calculus with de Bruijn indices* ($\lambda_{\mathtt{dB}}$ for short).

The sets of *terms* $\mathbb{T}_{\lambda_{\mathtt{dB}}}$ and *contexts* are given by the following grammar:

$$\textbf{Terms}\ t ::= \mathtt{i} \mid t\,t \mid \lambda t \qquad \textbf{Contexts}\ \mathtt{C} ::= \Box \mid \mathtt{C}\,t \mid t\,\mathtt{C} \mid \lambda\mathtt{C}$$

where $\mathtt{i} \in \mathbb{N}_{\geq 1}$ is called an *index*. Indices are place-holders indicating the distance to the binding abstraction. In the context of the $\lambda_{\mathtt{dB}}$-calculus, indices are also called *variables*. Thus, the *free variables* of a term are inductively defined as: $\mathsf{fv}(\mathtt{i}) \triangleq \{\mathtt{i}\}$; $\mathsf{fv}(t\,u) \triangleq \mathsf{fv}(t) \cup \mathsf{fv}(u)$; and $\mathsf{fv}(\lambda t) \triangleq \mathsf{fv}(t) - 1$, where $X - k$ stands for subtracting k from each element of the set X, removing those that result in a non-positive index.

In order to define β-reduction *à la* de Bruijn, the substitution of an index $\mathtt{i}$ for a term u in a term t must be defined. Therefore, it is necessary to identify among the indices of the term t, those corresponding to $\mathtt{i}$. Furthermore, the indices of u should be updated in order to preserve the correct bindings after the replacement of the variable by u. To that end, the *increment at depth* k for variables in a term t, written $\uparrow_k(t)$, is inductively defined as follows:

$$\uparrow_k(\mathtt{i}) \triangleq \begin{cases} \mathtt{i}+\mathtt{1} & \text{if } i > k \\ \mathtt{i} & \text{if } i \leq k \end{cases} \qquad \begin{aligned} \uparrow_k(t\,u) &\triangleq \uparrow_k(t)\uparrow_k(u) \\ \uparrow_k(\lambda t) &\triangleq \lambda\uparrow_{k+1}(t) \end{aligned}$$

Then, the *substitution at level* i of a term u in a term t, denoted $\{\mathtt{i} \setminus u\}t$, is defined as a partial function mapping free variables at level i to terms, performing the appropriate updates as it traverses the substituted term, to avoid undesired captures.

$$\{\mathtt{i} \setminus u\}\mathtt{i}' \triangleq \begin{cases} \mathtt{i}'-\mathtt{1} & \text{if } i' > i \\ u & \text{if } i' = i \\ \mathtt{i}' & \text{if } i' < i \end{cases} \qquad \begin{aligned} \{\mathtt{i} \setminus u\}(t\,s) &\triangleq \{\mathtt{i} \setminus u\}t\,\{\mathtt{i} \setminus u\}s \\ \{\mathtt{i} \setminus u\}\lambda t &\triangleq \lambda\{\mathtt{i}+\mathtt{1} \setminus \uparrow_0(u)\}t \end{aligned}$$

It is worth noticing that this substitution should be interpreted in the context of a redex, where a binder is removed and its bound index substituted. This forces to update the free indices, that might be captured by an outermost abstraction, as done by the first case of the substitution over a variable $\mathtt{i}'$. Hence, preserving the correct bindings.

Finally, the reduction relation $\to_{\mathtt{dB}}$ of the $\lambda_{\mathtt{dB}}$-calculus is given by the closure by contexts of the rewriting rule:

$$(\lambda s)\,u \mapsto_{\mathtt{dB}} \{\mathtt{1} \setminus u\}s$$

Also, embeddings between the λ-calculus and $\lambda_{\mathtt{dB}}$ are defined: $[\![\_]\!] : \mathbb{T}_\lambda \to \mathbb{T}_{\lambda_{\mathtt{dB}}}$

and $(\!|\_|\!) : \mathbb{T}_{\lambda_{\mathtt{dB}}} \to \mathbb{T}_{\lambda}$, in such a way that they are the inverse of each other and, they allow to simulate one calculus into the other:

Theorem 2.2 ([17]) *Let $t \in \mathbb{T}_{\lambda}$ and $s \in \mathbb{T}_{\lambda_{\mathtt{dB}}}$. Then,*

(i) *If $t \to_{\beta} t'$, then $[\![t]\!] \to_{\mathtt{dB}} [\![t']\!]$.*

(ii) *If $s \to_{\mathtt{dB}} s'$, then $(\!|s|\!) \to_{\beta} (\!|s'|\!)$.*

This shows that both formalisms (λ-calculus and $\lambda_{\mathtt{dB}}$) have exactly the same operational semantics.

As an example to illustrate both reduction in the $\lambda_{\mathtt{dB}}$-calculus and its equivalence with the λ-calculus, consider the following terms: $(\lambda z.\lambda y.z)\,(\lambda x.x)\,(\lambda x.x\,x)$ and $(\lambda\lambda\mathtt{2})\,(\lambda\mathtt{1})\,(\lambda\mathtt{1}\;\mathtt{1})$. The reader can verify that both expressions encode the same function in its respective calculus. As expected, their operational semantics coincide

$$\begin{array}{ccccc} (\lambda z.\lambda y.z)\,(\lambda x.x)\,(\lambda x.x\,x) & \to_{\beta} & (\lambda y.\lambda x.x)\,(\lambda x.x\,x) & \to_{\beta} & \lambda x.x \\ (\lambda\lambda\mathtt{2})\,(\lambda\mathtt{1})\,(\lambda\mathtt{1}\;\mathtt{1}) & \to_{\mathtt{dB}} & (\lambda\lambda\mathtt{1})\,(\lambda\mathtt{1}\;\mathtt{1}) & \to_{\mathtt{dB}} & \lambda\mathtt{1} \end{array}$$

3 The Pure Pattern Calculus with de Bruijn indices

This section introduces the novel *Pure Pattern Calculus with de Bruijn indices* ($\mathsf{PPC_{dB}}$). It represents a natural extension of de Bruijn ideas to a framework where a binder may capture more than one symbol. In the particular case of PPC there are two kinds of captured symbols, namely variables and matchables. This distinction is preserved in $\mathsf{PPC_{dB}}$ while extending indices to pairs (*a.k.a.* bidimensional indices) to distinguish the binder that captures the symbol and the individual symbol among all those captured by the same binder.

The sets of *terms* $\mathbb{T}_{\mathsf{PPC_{dB}}}$, *contexts*, *data structures* $\mathbb{D}_{\mathsf{PPC_{dB}}}$ and *matchable forms* $\mathbb{M}_{\mathsf{PPC_{dB}}}$ of $\mathsf{PPC_{dB}}$ are given by the following grammar:

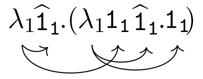
$$\begin{array}{llll} \textbf{Terms} & t ::= \mathtt{i_j} \mid \widehat{\mathtt{i_j}} \mid t\,t \mid \lambda_n t.t & \textbf{Data structures} & d ::= \widehat{\mathtt{i_j}} \mid d\,t \\ \textbf{Contexts} & \mathtt{C} ::= \Box \mid \mathtt{C}\,t \mid t\,\mathtt{C} \mid \lambda_n \mathtt{C}.t \mid \lambda_n t.\mathtt{C} & \textbf{Matchable forms} & m ::= d \mid \lambda_n t.t \end{array}$$

where $\mathtt{i_j}$ is dubbed a *bidimensional index* and denotes an ordered pair in $\mathbb{N}_{\geq 1} \times \mathbb{N}_{\geq 1}$ with *primary index* $\mathtt{i}$ and *secondary index* $\mathtt{j}$. The sub-index $n \in \mathbb{N}$ in an abstraction represents the amount of indices (pairs) being captured by it. The primary index of a pair is used to determine if the pair is bound by an abstraction, while the secondary index identifies the pair among those (possibly many) bound ones. As for PPC, an index of the form $\mathtt{i_j}$ is called a *variable index* while $\widehat{\mathtt{i_j}}$ is dubbed a *matchable index.* The *free variables* and *free matchables* of a term are thus defined as follows:

$$\begin{aligned}
\mathsf{fv}(\mathtt{i_j}) &\triangleq \{\mathtt{i_j}\} & \mathsf{fm}(\mathtt{i_j}) &\triangleq \emptyset\\
\mathsf{fv}(\widehat{\mathtt{i}}_\mathtt{j}) &\triangleq \emptyset & \mathsf{fm}(\widehat{\mathtt{i}}_\mathtt{j}) &\triangleq \{\mathtt{i_j}\}\\
\mathsf{fv}(t\,u) &\triangleq \mathsf{fv}(t) \cup \mathsf{fv}(u) & \mathsf{fm}(t\,u) &\triangleq \mathsf{fm}(t) \cup \mathsf{fm}(u)\\
\mathsf{fv}(\lambda_n p.t) &\triangleq \mathsf{fv}(p) \cup (\mathsf{fv}(t) - 1) & \mathsf{fm}(\lambda_n p.t) &\triangleq (\mathsf{fm}(p) - 1) \cup \mathsf{fm}(t)
\end{aligned}$$

where $X - k$ stands for subtracting k from the primary index of each element of the set X, removing those that result in a non-positive index.

Let us illustrate these concepts with a similar example as that given for PPC, namely the function $\mathsf{elim} = \lambda_{[x]}\widehat{x}.(\lambda_{[y]}x\,\widehat{y}.y)$. An equivalent term in the $\mathsf{PPC_{dB}}$ framework would be $\lambda_1\widehat{\mathtt{1}}_\mathtt{1}.(\lambda_1\mathtt{1_1}\,\widehat{\mathtt{1}}_\mathtt{1}.\mathtt{1_1})$. Note that variable indices in the context of a pattern are not bound by the respective abstraction, in the same way that matchable indices in the body of the abstraction are not captured either. Thus, the first occurrence of $\mathtt{1_1}$ is actually bound by the outermost abstraction, together with the first occurrence of the matchable index $\widehat{\mathtt{1}}_\mathtt{1}$. The rest of the indices in the term are bound by the inner abstraction as depicted in the figure to the right. As a further (more interesting) example, consider the term $\lambda_{[x,y]}\widehat{x}\,\widehat{y}.\lambda_{[]}x.y$ from PPC, whose counter-part in $\mathsf{PPC_{dB}}$ would look like $\lambda_2\widehat{\mathtt{1}}_\mathtt{1}\,\widehat{\mathtt{1}}_\mathtt{2}.\lambda_0\mathtt{1_1}.\mathtt{2_2}$. This example illustrates the use of secondary indices to identify symbols bound by the same abstraction. It also shows how the primary index of a variable is increased when occurring within the body of an internal abstraction, while this is not the case for occurrences in a pattern position. Thus, both $\mathtt{1_1}$ and $\mathtt{2_2}$ are bound by the outermost abstraction, as well as $\widehat{\mathtt{1}}_\mathtt{1}$ and $\widehat{\mathtt{1}}_\mathtt{2}$. Note how the inner abstraction does not bind any index at all.



A term t is said to be *well-formed* if all the free bidimensional indices (variables and matchables) of t have their secondary index equal to 1, and for every sub-term of the form $\lambda_n p.s$ (written $\lambda_n p.s \subseteq t$) all the pairs captured by the abstraction have their secondary index within the range $[1, n]$. Formally, $\{\mathtt{i_j} \mid \mathtt{i_j} \in \mathsf{fm}(t) \cup \mathsf{fv}(t), j > 1\} \cup (\bigcup_{\lambda_n p.s \subseteq t} \{\mathtt{1_j} \mid \mathtt{1_j} \in \mathsf{fm}(p) \cup \mathsf{fv}(s), j > n\}) = \emptyset$.

Before introducing a proper notion of substitution for $\mathsf{PPC_{dB}}$ it is necessary to have a mechanism to update indices at a certain depth within the term. The *increment at depth k* for variable and matchable indices in a term t, written $\uparrow_k(t)$ and $\Uparrow_k(t)$ respectively, are inductively defined as follows

$$\begin{aligned}
\uparrow_k(\mathtt{i_j}) &\triangleq \begin{cases}(\mathtt{i}+\mathtt{1})_\mathtt{j} & \text{if } i > k\\ \mathtt{i_j} & \text{if } i \leq k\end{cases} & \Uparrow_k(\mathtt{i_j}) &\triangleq \mathtt{i_j}\\
\uparrow_k(\widehat{\mathtt{i}}_\mathtt{j}) &\triangleq \widehat{\mathtt{i}}_\mathtt{j} & \Uparrow_k(\widehat{\mathtt{i}}_\mathtt{j}) &\triangleq \begin{cases}\widehat{(\mathtt{i}+\mathtt{1})}_\mathtt{j} & \text{if } i > k\\ \widehat{\mathtt{i}}_\mathtt{j} & \text{if } i \leq k\end{cases}\\
\uparrow_k(t\,u) &\triangleq \uparrow_k(t)\uparrow_k(u) & \Uparrow_k(t\,u) &\triangleq \Uparrow_k(t)\Uparrow_k(u)\\
\uparrow_k(\lambda_n p.t) &\triangleq \lambda_n\uparrow_k(p).\uparrow_{k+1}(t) & \Uparrow_k(\lambda_n p.t) &\triangleq \lambda_n\Uparrow_{k+1}(p).\Uparrow_k(t)
\end{aligned}$$

Similarly, the *decrement at depth k* for variables ($\downarrow_k(\_)$) and matchables ($\Downarrow_k(\_)$) are defined by subtracting one from the primary index above k in the term. Most of the times these functions are used with $k = 0$, thus the subindex will be omitted when it is clear from context. In particular, the decrement function for variables will allow us to generalise the idea of substitution at level i with respect to the original one presented in Sec. 2.2, which only holds in the context of a β-reduction, by making the necessary adjustments to the indices at the moment of the redution instead of hard-coding them into the substitution meta-operation.

A *substitution at level i* is a partial function from variable indices to terms. It maps free variable indices at level i to terms, performing the appropriate updates as it traverses the substituted term, to avoid undesired captures.

$$\{\mathtt{i}_\mathtt{j} \setminus u_j\}_{j\in J}\mathtt{i'}_\mathtt{k} \triangleq \begin{cases} u_k & \text{if } i' = i, k \in J \\ \mathtt{i'}_\mathtt{k} & \text{if } i' \not= i \end{cases} \qquad \{\mathtt{i}_\mathtt{j} \setminus u_j\}_{j\in J}(t\,s) \triangleq \{\mathtt{i}_\mathtt{j} \setminus u_j\}_{j\in J}t\,\{\mathtt{i}_\mathtt{j} \setminus u_j\}_{j\in J}s$$

$$\{\mathtt{i}_\mathtt{j} \setminus u_j\}_{j\in J}\widehat{\mathtt{i'}}_\mathtt{k} \triangleq \widehat{\mathtt{i'}}_\mathtt{k} \qquad \{\mathtt{i}_\mathtt{j} \setminus u_j\}_{j\in J}\lambda_n p.t \triangleq \lambda_n\{\mathtt{i}_\mathtt{j} \setminus \Uparrow(u_j)\}_{j\in J}p.\{(\mathtt{i}+\mathtt{1})_\mathtt{j} \setminus \uparrow(u_j)\}_{j\in J}$$

It is worth noticing that the base case for variable indices is undefined if $i' = i$ and $k \notin J$. Such case will render the result of the substitution undefined as well. In the operational semantics of $\mathsf{PPC_{dB}}$, the matching operation presented below will be responsible for avoiding this undesired situation, as we will see later. The *domain* of a substitution at level i is given by $\mathsf{dom}(\{\mathtt{i}_\mathtt{j} \setminus u_j\}_{j\in J}) \triangleq \{\mathtt{i}_\mathtt{j}\}_{j\in J}$. The identity substitution (*i.e.* with empty domain) is denoted $\{\}$ or id.

As in PPC, a match ($\mu, \nu, \ldots$) may succeed, fail ($\mathtt{fail}$) or be undetermined ($\mathtt{wait}$). For $\mathsf{PPC_{dB}}$, a successful match will yield a substitution at level 1, as given by the following *matching operation*, where the rules are applied in order as in PPC:

$$\begin{array}{rcll} \{\!\{\widehat{\mathtt{1}}_\mathtt{j} \setminus^n u\}\!\} & \triangleq & \{\mathtt{1}_\mathtt{j} \setminus u\} & \\ \{\!\{\widehat{\mathtt{i}+\mathtt{1}}_\mathtt{j} \setminus^n \widehat{\mathtt{i}}_\mathtt{j}\}\!\} & \triangleq & \{\} & \\ \{\!\{p\,q \setminus^n t\,u\}\!\} & \triangleq & \{\!\{p \setminus^n t\}\!\} \uplus \{\!\{q \setminus^n u\}\!\} & \text{if } t\,u, p\,q \in \mathbb{M}_{\mathsf{PPC_{dB}}} \\ \{\!\{p \setminus^n u\}\!\} & \triangleq & \mathtt{fail} & \text{if } u, p \in \mathbb{M}_{\mathsf{PPC_{dB}}} \\ \{\!\{p \setminus^n u\}\!\} & \triangleq & \mathtt{wait} & \text{otherwise} \end{array}$$

where disjoint union of matching is adapted to $\mathsf{PPC_{dB}}$ from PPC in a straightforward way. The first two rules in the matching operation for $\mathsf{PPC_{dB}}$ are worth a comment. As the matching operation should be understood in the context of a redex, the matchable symbols bound in the pattern are those with primary index equal to 1. Thus, $\mathsf{PPC_{dB}}$'s counter-part of the membership check $x \in \theta$ from PPC's matching operation is a simple syntactic check on the primary index. Similarly, $x \notin \theta$ corresponds to the primary index being greater than 1, as checked by the second rule of the definition. However, a primary index $\widehat{\mathtt{i}+\mathtt{1}}$ within the pattern should match primary index $\widehat{\mathtt{i}}$ from the argument, since the former is affected by an extra binder in a redex. For instance, in the term $(\lambda_1\widehat{\mathtt{2}}_\mathtt{1}\,\widehat{\mathtt{1}}_\mathtt{1}.\mathtt{1}_\mathtt{1})\,(\widehat{\mathtt{1}}_\mathtt{1}\,t')$ the matchable $\widehat{\mathtt{2}}_\mathtt{1}$ is free and corresponds to $\widehat{\mathtt{1}}_1$ in the argument, while $\widehat{\mathtt{1}}_1$ from the pattern is bound by the abstraction. Its counter-part in PPC would be α-equivalent to $(\lambda_{[x]}\widehat{y}\,\widehat{x}.x)\,(\widehat{y}\,t)$. Hence, $\{\!\{\widehat{\mathtt{2}}_1\,\widehat{\mathtt{1}}_1 \setminus^1 \widehat{\mathtt{1}}_1\,t'\}\!\} = \{\mathtt{1}_1 \setminus t'\}$.

As for PPC, an additional post-condition is checked over $\{\!\{p \setminus^n u\}\!\}$ to prevent indices from going out of scope. It requires $\mathsf{dom}(\{\!\{p \setminus^n u\}\!\}) = \{\mathtt{1}_1, \ldots, \mathtt{1}_\mathtt{n}\}$, which essentially implies that all the bound indices are assigned a value by the resulting substitution. This condition can be guaranteed by requesting $\{\mathtt{1}_1, \ldots, \mathtt{1}_\mathtt{n}\} \subseteq \mathsf{fm}(p)$, for each abstraction $\lambda_n p.t$ within a well-formed term. To illustrate the need of such a check, consider the term $(\lambda_2\widehat{\mathtt{1}}_1.\mathtt{1}_2)\,u'$ (*i.e.* the $\mathsf{PPC_{dB}}$ counter-part of $(\lambda_{[x,y]}\widehat{x}.y)\,u$, given in Sec. 2.1). If the matching $\{\!\{\widehat{\mathtt{1}}_1 \setminus^2 u'\}\!\} = \{\mathtt{1}_1 \setminus u'\}$ is considered correct, then no replacement for the variable index $\mathtt{1}_2$ in the body of the abstraction is set, resulting in an ill-behaved operational semantics.

The reduction relation $\to_{\mathtt{dB}}$ of $\mathsf{PPC_{dB}}$ is given by the closure by contexts of the rewriting rule:

$$(\lambda_n p.s)\,u \mapsto_{\mathtt{dB}} \downarrow(\{\!\{p \setminus^n \uparrow(u)\}\!\}\,s)$$

whenever $\{\!\{p\setminus^n\uparrow(u)\}\!\}$ is a decided match. The decrement function for variable indices is applied to the reduct to compensate for the loss of a binder over s. However, the variable indices of u are not affected by such binder in the redex. Hence the need of incrementing them prior to the (eventual) substitution.

Following the reduction example given above for PPC, consider these codifications of $\mathsf{elim} = \lambda_{[x]}\widehat{x}.(\lambda_{[y]}x\,\widehat{y}.y)$ and $\lambda_{[z]}\widehat{z}.\widehat{c}\,z\,\widehat{n}$ respectively: $\lambda_1\widehat{\mathtt{1}}_1.(\lambda_1\mathtt{1}_1\,\widehat{\mathtt{1}}_1.\mathtt{1}_1)$ and $\lambda_1\widehat{\mathtt{1}}_1.\widehat{\mathtt{1}}_1\,\mathtt{1}_1\,\widehat{\mathtt{2}}_1$. Note how the first occurrence of $\mathtt{1}_1$ is actually bound by the outermost abstraction, since abstractions do not bind variable indices in their pattern. Similarly, the matchable index $\widehat{\mathtt{1}}_1$ in the body of $\widehat{\lambda}_1\widehat{\mathtt{1}}_1.\mathtt{1}_1\,\widehat{\mathtt{1}}_1\,2_1$ turns out to be free as well as $\widehat{\mathtt{2}}_1$. Then, as expected, we have the following sequence:

$$(\lambda_1\widehat{\mathtt{1}}_1.(\lambda_1\mathtt{1}_1\,\widehat{\mathtt{1}}_1.\mathtt{1}_1))\,(\lambda_1\widehat{\mathtt{1}}_1.\widehat{\mathtt{1}}_1\,\mathtt{1}_1\,\widehat{\mathtt{2}}_1) \to_{\mathtt{dB}} \quad \lambda_1(\lambda_1\widehat{\mathtt{1}}_1.\widehat{\mathtt{2}}_1\,\mathtt{1}_1\,\widehat{\mathtt{3}}_1)\,\widehat{\mathtt{1}}_1.\mathtt{1}_1 \to_{\mathtt{dB}} \quad \lambda_1\widehat{\mathtt{2}}_1\,\widehat{\mathtt{1}}_1\,\widehat{\mathtt{3}}_1.\mathtt{1}_1$$

In the first step, $\lambda_1\widehat{\mathtt{1}}_1.\widehat{\mathtt{1}}_1\,\mathtt{1}_1\,\widehat{\mathtt{2}}_1$ is substituted for $\mathtt{1}_1$ into the pattern $\mathtt{1}_1\,\widehat{\mathtt{1}}_1$. The fact that the substitution takes place within the context of a pattern forces the application of $\Uparrow(\_)$, thus updating the matchable indices and obtaining $\lambda_1\widehat{\mathtt{1}}_1.\widehat{\mathtt{2}}_1\,\mathtt{1}_1\,\widehat{\mathtt{3}}_1$. Note that the increment and decrement added by the reduccion rule take no effect as there are no free variable indices in the term. In the second step, the resulting application is reduced, giving place to a term whose counter-part in PPC would be equivalent to $\lambda_{[y]}\widehat{c}\,\widehat{y}\,\widehat{n}.y$ (*cf.* the reduction example in Sec. 2.1).

In the following sections $\mathsf{PPC_{dB}}$ is shown to be equivalent to PPC in terms of expressive power and operational semantics. The main advantage of this new presentation is that it gets rid of α-conversion, since there is no possible collision between free and bound variables/matchables. However, there is one minor drawback with respect to the use of de Bruijn indices for the standard λ-calculus. As mentioned above, when working with de Bruijn indices in the standard λ-calculus, α-equivalence becomes syntactical equality.

Unfortunately, this is not the case when working with bidimensional indices. For instance, consider the terms $\lambda_2\widehat{1}_1\,\widehat{1}_2.1_1$ and $\lambda_2\widehat{1}_2\,\widehat{1}_1.1_2$. Both represent the function that decomposes an application and projects its first component. But they differ in the way the secondary indices are assigned. Moreover, one may be tempted to impose an order for the way the secondary indices are assigned within the pattern to avoid this situation (recall that the post-condition of the matching operation forces all bound symbols to appear in the pattern). Given the dynamic nature of patterns in the PPC framework, this enforcement would not solve the problem since patterns may reduce and such an order is not closed under reduction. For example, consider $\lambda_2(\lambda_2\widehat{1}_1\,\widehat{1}_2.1_2\,1_1)\,(\widehat{1}_1\,\widehat{1}_2).1_1 \rightarrow_{\mathsf{dB}} \lambda_2\widehat{1}_2\,\widehat{1}_1.1_1$.

Fortunately enough, this does not represent a problem from the implementation point of view, since the ambiguity is local to a binder and does not imply the need for "renaming" variables/matchables while reducing a term, *i.e.* no possible undesired capture can happen because of it. It is important to note though, that in the sequel, when refering to equality over terms of $\mathsf{PPC_{dB}}$, it is not syntactical equality but equality modulo these assignments for secondary indices that we are using.

4 Translation

This section introduces translations between PPC and $\mathsf{PPC_{dB}}$ (back and forth). The goal is to show that these interpretations are suitable to simulate one calculus into the other. Moreover, the proposed translations turn out to be the inverse of each other (modulo α-conversion) and, as we will see in Sec. 5, they allow to formalise a strong bisimulation between the two calculi.

We start with the translation from PPC to $\mathsf{PPC_{dB}}$. It takes the term to be translated together with two lists of lists of symbols that dictate how the variables and matchables of the terms should be interpreted respectively. We use lists of lists since the first dimension indicates the distance to the binder, while the second identifies the symbol among the multiple bound ones.

Given the lists of lists X and Y, we denote by XY their concatenation. To improve readability, when it is clear from context, we also write θX with θ a list of symbols to denote $[\theta]X$ where $[\_]$ denotes the list constructor. We use set operations like union and intersection over lists to denote the union/intersection of its underlying sets.

Definition 4.1 Given a term $t \in \mathbb{T}_{\mathsf{PPC}}$ and lists of lists of symbols V and M such that $\mathsf{fv}(t) \subseteq \bigcup_{V' \in V} V'$ and $\mathsf{fm}(t) \subseteq \bigcup_{M' \in M} M'$, the *translation of t relative to V and M*, written $[\![t]\!]_V^M$, is inductively defined as follows:

$$\begin{aligned}
[\![x]\!]_V^M &\triangleq \mathtt{i_j} && \text{where } i = \min\{i' \mid x \in V_{i'}\} \text{ and } j = \min\{j' \mid x = V_{ij'}\}\\
[\![\widehat{x}]\!]_V^M &\triangleq \widehat{\mathtt{i}}_{\mathtt{j}} && \text{where } i = \min\{i' \mid x \in M_{i'}\} \text{ and } j = \min\{j' \mid x = M_{ij'}\}\\
[\![t\,u]\!]_V^M &\triangleq [\![t]\!]_V^M\,[\![u]\!]_V^M &&\\
[\![\lambda_\theta p.t]\!]_V^M &\triangleq \lambda_{|\theta|}[\![p]\!]_V^{\theta M}.[\![t]\!]_{\theta V}^M &&
\end{aligned}$$

Let $x_1, x_2, \ldots$ be an enumeration of $\mathbb{V}$. Then, the *translation* of t to $\mathsf{PPC_{dB}}$, written simply $[\![t]\!]$, is defined as $[\![t]\!]^X_X$ where $X = [[x_1], \ldots, [x_n]]$ such that $\mathsf{fv}(t) \cup \mathsf{fm}(t) \subseteq \{x_1, \ldots, x_n\}$.

For example, consider the term $s_0 = (\lambda_{[x]}\widehat{y}\,\widehat{x}.x)\,(\widehat{y}\,s'_0)$ with $\mathsf{fv}(s'_0) = \{y\}$ and $\mathsf{fm}(s'_0) = \{y\}$. Then, $[\![s_0]\!]^{[[y]]}_{[[y]]} = (\lambda_1 [\![\widehat{y}]\!]^{[[x],[y]]}_{[[y]]}\,[\![\widehat{x}]\!]^{[[x],[y]]}_{[[y]]}.[\![x]\!]^{[[y]]}_{[[x],[y]]})\,([\![\widehat{y}]\!]^{[[y]]}_{[[y]]}\,[\![s'_0]\!]^{[[y]]}_{[[y]]}) = (\lambda_1 \widehat{\mathsf{2}}_1\,\widehat{\mathsf{1}}_1.\mathsf{1}_1)\,(\widehat{\mathsf{1}}_1\,[\![s'_0]\!]^{[[y]]}_{[[y]]})$. Note that the inicialisation of V and M with singleton elements implies that each free variable/matchable in the term will be assigned a distinct primary index (when interpreted at the same depth), following de Bruijn's original ideas: let $s_1 = (\lambda_{[x]}\widehat{y}\,\widehat{x}.x)\,(\widehat{y}\,z)$, then $[\![s_1]\!]^{[[y],[z]]}_{[[y],[z]]} = (\lambda_1 \widehat{\mathsf{2}}_1\,\widehat{\mathsf{1}}_1.\mathsf{1}_1)\,(\widehat{\mathsf{1}}_1\,\mathsf{2}_1)$.

Our main goal is to prove that $\mathsf{PPC_{dB}}$ simulates PPC via this embedding. For this purpose we need to state first some auxiliary lemmas that prove how the translation behaves with respect to the substitution and the matching operation. We start with a technical result concerning the increment functions for variable and matchable indices. Notation $\uparrow^n_k(t)$ stands for n consecutive applications of $\uparrow_k(\_)$ over t (similarly for $\Uparrow^n_k(t)$).

Lemma 4.2 *Let $t \in \mathbb{T}_{\mathsf{PPC}}$, $k \geq 0$, $i \geq 1$ and $n \geq k+i$ such that $X_l \cap (\mathsf{fv}(t) \cup \mathsf{fm}(t)) = \emptyset$ for all $l \in [k+1, k+i-1]$. Then,*

(i) $[\![t]\!]^M_{X_1 \ldots X_n} = \uparrow^{i-1}_k([\![t]\!]^M_{X_1 \ldots X_k X_{k+i} \ldots X_n})$.

(ii) $[\![t]\!]^{X_1 \ldots X_n}_V = \Uparrow^{i-1}_k([\![t]\!]^{X_1 \ldots X_k X_{k+i} \ldots X_n}_V)$.

The translation of a substitution σ requires an enumeration θ such that $\mathsf{dom}(\sigma) \subseteq \theta$ to be provided. It is then defined as $[\![\sigma, \theta]\!]^M_V \triangleq \{\mathsf{1_j} \setminus [\![\sigma x_j]\!]^M_V\}_{x_j \in \mathsf{dom}(\sigma)}$. Note how substitutions from PPC are mapped into substitutions at level 1 in the $\mathsf{PPC_{dB}}$ framework. This suffices since substitutions are only meant to be created in the context of a redex. When acting at arbitrary depth on a term, substitutions are shown to behave properly.

Lemma 4.3 *Let $s \in \mathbb{T}_{\mathsf{PPC}}$, σ be a substitution, θ be an enumeration such that $\mathsf{dom}(\sigma) \subseteq \theta$ and Y be a list of $i-1$ lists of symbols such that $(\bigcup_{Y' \in Y} Y') \cap \theta = \emptyset$ and $(\bigcup_{Y' \in Y} Y') \cap (\bigcup_{x_j \in \theta} \mathsf{fv}(\sigma x_j)) = \emptyset$.*
Then, $[\![\sigma s]\!]^M_{YX} = \downarrow_{i-1}(\{\mathtt{i_j} \setminus \uparrow_{i-1}([\![\sigma x_j]\!]^M_{YX})\}_{x_j \in \mathsf{dom}(\sigma)} [\![s]\!]^M_{Y\theta X})$.

In the case of a match, its translation is given by $[\![\{\!\{p \setminus^{\theta} u\}\!\}]\!]^M_V \triangleq \{\!\{[\![p]\!]^{\theta M}_V \setminus^{|\theta|} [\![u]\!]^M_V\}\!\}$. Note how θ is pushed into the matchable symbol list of the pattern, in accordance with the translation of an abstraction. This is crucial for the following result of preservation of the matching output.

Lemma 4.4 *Let $p, u \in \mathbb{T}_{\mathsf{PPC}}$.*

(i) *If $\{\!\{p \setminus^{\theta} u\}\!\} = \sigma$, then $[\![\{\!\{p \setminus^{\theta} u\}\!\}]\!]^M_V = [\![\sigma, \theta]\!]^M_V$.*

(ii) *If $\{\!\{p \setminus^{\theta} u\}\!\} = \mathtt{fail}$, then $[\![\{\!\{p \setminus^{\theta} u\}\!\}]\!]^M_V = \mathtt{fail}$.*

(iii) *If $\{\!\{p \setminus^{\theta} u\}\!\} = \mathtt{wait}$, then $[\![\{\!\{p \setminus^{\theta} u\}\!\}]\!]^M_V = \mathtt{wait}$.*

These previous results will allow to prove the simulation of PPC into $\mathsf{PPC_{dB}}$ via the translation $[\![\_]\!]$. We postpone this result to Sec. 5 (*cf.* Thm. 5.1).

Now we focus on the converse side of the embedding, *i.e.* translation of $\mathsf{PPC_{dB}}$ terms into PPC terms. As before, this mapping requires two lists of lists of symbols from which names of the free indices of the term will be selected: one for variable indices and the other for matchable indices.

Definition 4.5 Given a term $t \in \mathbb{T}_{\mathsf{PPC_{dB}}}$ and lists of lists of distinct symbols V and M such that V_{ij} is defined for every $\mathtt{i_j} \in \mathsf{fv}(t)$ and M_{ij} is defined for every $\mathtt{i_j} \in \mathsf{fm}(t)$, the *translation of t relative to V and M*, written $(\!|t|\!)^M_V$, is inductively defined as follows:

$$\begin{aligned}
(\!|\mathtt{i_j}|\!)^M_V &\triangleq V_{ij}\\
(\!|\widehat{\mathtt{i_j}}|\!)^M_V &\triangleq \widehat{M_{ij}}\\
(\!|t\,u|\!)^M_V &\triangleq (\!|t|\!)^M_V\,(\!|u|\!)^M_V\\
(\!|\lambda_n p.t|\!)^M_V &\triangleq \lambda_\theta (\!|p|\!)^{\theta M}_V . (\!|t|\!)^M_{\theta V} \quad \theta = [x_1,\ldots,x_n] \text{ fresh symbols}
\end{aligned}$$

Let $x_1, x_2, \ldots$ be the same enumeration of $\mathbb{V}$ as in Def. 4.1. Then, the *translation* of t to PPC, written simply $(\!|t|\!)$, is defined as $(\!|t|\!)^X_X$ where $X = [[x_1],\ldots,[x_n]]$ such that $\mathsf{fv}(t) \cup \mathsf{fm}(t) \subseteq \{\mathtt{1_1},\ldots,\mathtt{n_1}\}$. Note that well-formedness of terms guarantees that X satisfies the conditions above.

To illustrate the translation, consider the term $t_1 = (\lambda_1 \widehat{\mathtt{2_1}}\,\widehat{\mathtt{1_1}}.\mathtt{1_1})\,(\widehat{\mathtt{1_1}}\,\mathtt{2_1})$ where $\mathsf{fv}(t_1) = \{\mathtt{2_1}\}$ and $\mathsf{fm}(t_1) = \{\mathtt{1_1}\}$. Then,
$(\!|t_1|\!)^{[[y],[z]]}_{[[y],[z]]} = (\lambda_{[x]} (\!|\widehat{\mathtt{2_1}}\,\widehat{\mathtt{1_1}}|\!)^{[[x],[y],[z]]}_{[[y],[z]]} . (\!|\mathtt{1_1}|\!)^{[[y],[z]]}_{[[x],[y],[z]]})\,(\!|\widehat{\mathtt{1_1}}\,\mathtt{2_1}|\!)^{[[y],[z]]}_{[[y],[z]]} = (\lambda_{[x]} \widehat{y}\,\widehat{x}.x)\,(\widehat{y}\,z)$. Note that $t_1 = [\![s_1]\!]$ from the example after Def. 4.1 and, with a proper initialisation of the lists V and M, we get $(\!|t_1|\!) = s_1$.

Once again, we start with some technical lemmas for substitutions and the matching operations with respect to the embedding $(\!|\_|\!)$. In this case, the increment functions for variable and matchable indices behave as follows:

Lemma 4.6 *Let $t \in \mathbb{T}_{\mathsf{PPC_{dB}}}$, $k \geq 0$, $i \geq 1$ and $n \geq k+i$ such that $\mathsf{fv}(t) \cup \mathsf{fm}(t) \subseteq \{\mathtt{i'_j} \mid i' \leq n-(i-1), j \leq |X_{i'}|\}$. Then,*

(i) $(\!|\uparrow^{i-1}_k(t)|\!)^M_{X_1\ldots X_n} =_\alpha (\!|t|\!)^M_{X_1\ldots X_k X_{k+i}\ldots X_n}$.

(ii) $(\!|\Uparrow^{i-1}_k(t)|\!)^{X_1\ldots X_n}_V =_\alpha (\!|t|\!)^{X_1\ldots X_k X_{k+i}\ldots X_n}_V$.

As for the converse, the translation is only defined for substitution at level 1 and requires to be provided a list of symbols θ such that $|\theta| \geq \max\{j \mid \mathtt{1_j} \in \mathsf{dom}(\sigma)\}$. Then, $(\!|\sigma,\theta|\!)^M_V \triangleq \{\theta_j \setminus (\!|\sigma\mathtt{1_j}|\!)^M_V\}_{\mathtt{1_j}\in\mathsf{dom}(\sigma)}$. The application of a substitution at an arbitrary level i is shown to translate properly.

Lemma 4.7 *Let $s \in \mathbb{T}_{\mathsf{PPC_{dB}}}$, σ be a substitution at level i, θ be a list of fresh symbols such that $|\theta| \geq \max\{j \mid \mathtt{i_j} \in \mathsf{dom}(\sigma)\}$ and Y be a list of $i-1$ lists of symbols. Then,* $(\!|\downarrow_{i-1}(\{\mathtt{i_j} \setminus \uparrow_{i-1}(\sigma\mathtt{i_j})\}_{\mathtt{i_j}\in\mathsf{dom}(\sigma)} s)|\!)^M_{YX} =_\alpha \{\theta_j \setminus (\!|\sigma\mathtt{i_j}|\!)^M_{YX}\}_{\mathtt{i_j}\in\mathsf{dom}(\sigma)} (\!|s|\!)^M_{Y\theta X}$.

Similarly to the substitution case, the translation of a match $\{\!\{p \backslash^n u\}\!\}$ requires to be supplied with a list of n fresh symbols θ. Then, it is defined as $(\!|\{\!\{p \backslash^n u\}\!\}, \theta|\!)^M_V \triangleq \{\!\{(\!|p|\!)^{\theta M}_V \backslash^\theta (\!|u|\!)^M_V\}\!\}$. The newly provided list of symbols is used both as the parameter of the resulting match and to properly translate the pattern, obtaining the following expected result.

Lemma 4.8 *Let $p, u \in \mathbb{T}_{\mathsf{PPC_{dB}}}$.*

(i) *If $\{\!\{p \backslash^n u\}\!\} = \sigma$, then $(\!|\{\!\{p \backslash^n u\}\!\}, \theta|\!)^M_V = (\!|\sigma, \theta|\!)^M_V$.*

(ii) *If $\{\!\{p \backslash^n u\}\!\} = \mathtt{fail}$, then $(\!|\{\!\{p \backslash^n u\}\!\}, \theta|\!)^M_V = \mathtt{fail}$.*

(iii) *If $\{\!\{p \backslash^n u\}\!\} = \mathtt{wait}$, then $(\!|\{\!\{p \backslash^n u\}\!\}, \theta|\!)^M_V = \mathtt{wait}$.*

Now we are in conditions to prove the simulation of $\mathsf{PPC_{dB}}$ into PPC via the translation provided in Def. 4.5.

Before proceeding to the next section, one final result concerns the translations. It turns out that each translation is the inverse of the other, as shown in Thm. 4.9. In case of PPC terms we should work modulo α-conversion, while for $\mathsf{PPC_{dB}}$ terms we may use equality (modulo secondary indices permutations, *cf.* last paragraph in Sec. 3). This constitutes the main result of this section and is the key to extend our individual simulation results (*cf.* Thm. 5.1 and 5.2 resp.) into a strong bisimulation between the two calculi, as shown in Sec. 5.

Theorem 4.9 (Invertibility) *Let $t \in \mathbb{T}_{\mathsf{PPC_{dB}}}$ and $s \in \mathbb{T}_{\mathsf{PPC}}$. Then,* (i) $[\![(\!|t|\!)]\!] = t$; *and* (ii) $(\!|[\![s]\!]|\!) =_\alpha s$.

Proof By straightforward induction on t and s respectively. Details in [19]. □

5 Strong bisimulation

In this section we prove the simulation of one calculus by the other via the proper translation and, most importantly, the strong bisimulation that follows after the invertibility result (*cf.* Thm. 4.9). This strong bisimulation result will allow to port many important properties already known for PPC into $\mathsf{PPC_{dB}}$, as we will discuss later.

We start by simulating PPC by $\mathsf{PPC_{dB}}$. The key step here is the preservation of the matching operation shown for $[\![\_]\!]$ in Lem. 4.4. It guarantees that every redex in PPC turns into a redex in $\mathsf{PPC_{dB}}$ too. Then, the appropiate definition of the operational semantics given for $\mathsf{PPC_{dB}}$ in Sec. 3 allows us to conclude.

Theorem 5.1 *Let $t \in \mathbb{T}_{\mathsf{PPC}}$. If $t \to_{\mathsf{PPC}} t'$, then $[\![t]\!] \to_{\mathsf{dB}} [\![t']\!]$.*

Proof By induction on $t \to_{\mathsf{PPC}} t'$ using Lem. 4.2, 4.3 and 4.4. Details in [19]. □

Regarding the converse simulation, *i.e.* $\mathsf{PPC_{dB}}$ into PPC, we resort here to the fact that the embedding $(\!|\_|\!)$ also preserves the matching operation (*cf.* Lem. 4.8). Then, every redex in $\mathsf{PPC_{dB}}$ is translated into a redex in PPC as well.

Theorem 5.2 *Let $t \in \mathbb{T}_{\mathsf{PPC_{dB}}}$. If $t \to_{\mathsf{dB}} t'$, then $(\!| t |\!) \to_{\mathsf{PPC}} (\!| t' |\!)$.*

Proof By induction on $t \to_{\mathsf{dB}} t'$ using Lem. 4.6, 4.7 and 4.8. Details in [19]. $\square$

As already commented, these previous results may be combined to obtain a strong bisimulation between the two calculi. The invertibility result allows to define a relation between terms in PPC and $\mathsf{PPC_{dB}}$. Given $t \in \mathbb{T}_{\mathsf{PPC_{dB}}}$ and $s \in \mathbb{T}_{\mathsf{PPC}}$, let us write $t \mapsto\!\!\Rightarrow s$ whenever $(\!| t |\!) =_\alpha s$ and, therefore, $[\![s]\!] = t$ by Thm. 4.9. Then, the strong bisimulation result states that whenever $t \mapsto\!\!\Rightarrow s$ and $t \to_{\mathsf{dB}} t'$, there exists a term s' such that $t' \mapsto\!\!\Rightarrow s'$ and $s \to_{\mathsf{PPC}} s'$, and the other way around. Graphically:

$$\begin{tikzcd}
t \arrow[r, Rightarrow, maps to] \arrow[d, "\mathsf{dB}"'] & s \arrow[d, dashed, "\mathsf{PPC}"'] & \text{and} & t \arrow[r, Rightarrow, maps to] \arrow[d, dashed, "\mathsf{dB}"'] & s \arrow[d, "\mathsf{PPC}"'] \\
t' \arrow[r, Rightarrow, maps to] & s' & & t' \arrow[r, Rightarrow, maps to] & s'
\end{tikzcd}$$

Theorem 5.3 (Strong bisimulation) *The relation $\mapsto\!\!\Rightarrow$ is a strong bisimulation with respect to the reduction relations $\to_{\mathsf{PPC}}$ and $\to_{\mathsf{dB}}$ respectively.*

Proof The proof follows immediately from Thm. 5.1 and 5.2 above, and the invertibility result (Thm. 4.9) given in Sec. 4. $\square$

The importance of this result resides in the fact that it guarantees that PPC and $\mathsf{PPC_{dB}}$ have exactly the same operational semantics. An immediate consequence of this is the confluence of $\to_{\mathsf{dB}}$.

Theorem 5.4 (Confluence) *The reduction relation $\to_{\mathsf{dB}}$ is confluent (CR).*

Proof The result follows directly from Thm. 2.1 and the strong bisimulation:

$$\begin{tikzcd}
 & & t \arrow[lld, twoheadrightarrow, "\mathsf{dB}"'] \arrow[rrd, twoheadrightarrow, "\mathsf{dB}"] \arrow[d, Rightarrow, maps to] & & \\
 & & s \arrow[ld, dashed, twoheadrightarrow, "\mathsf{PPC}"'] \arrow[rd, dashed, twoheadrightarrow, "\mathsf{PPC}"] & & \\
t_0 \arrow[r, Rightarrow, maps to] \arrow[rrdd, dashed, twoheadrightarrow, "\mathsf{dB}"'] & s_0 \arrow[rd, dashed, twoheadrightarrow, "\mathsf{PPC}"'] & \text{Thm. 2.1} & s_1 \arrow[ld, dashed, twoheadrightarrow, "\mathsf{PPC}"] & t_1 \arrow[l, Rightarrow, maps to] \arrow[lldd, dashed, twoheadrightarrow, "\mathsf{dB}"] \\
 & & s' & & \\
 & & t' \arrow[u, Rightarrow, maps to] & &
\end{tikzcd}$$

$\square$

Another result of relevance to our line of research, is the existence of normalising reduction strategies for PPC, as it is shown in [4]. This result is particularly challenging since reduction in PPC is shown to be *non-sequential* due to the nature of its matching operation. This implies that the notion of *needed redexes* (a key concept for defining normalising strategies) must be generalised to *necessary sets* [26] of redexes. Moreover, the notion of *gripping* [21] is also captured by PPC, representing a further obstacle in the definition of such a normalising strategy. All in all, in [4] the authors introduce a reduction strategy $\mathcal{S}$ that is shown to be normalising, overcoming all the aforementioned issues. Thanks to the strong bisimulation result presented above, this strategy $\mathcal{S}$ can also be guaranteed to normalise for $\mathsf{PPC_{dB}}$.

6 Conclusion

In this paper we introduced a novel presentation of the *Pure Pattern Calculus* (PPC) [15] in de Bruijn's style. This required extending de Bruijn ideas for a setting where each binder may capture more than one variable at once. To this purpose we defined *bidimensional indices* of the form $\mathtt{i_j}$ where $\mathtt{i}$ is dubbed the *primary index* and $\mathtt{j}$ the *secondary index*, so that the primary index determines the binding abstraction and the secondary index identifies the variable among those (possibly many) bound ones. Moreover, given the nature of PPC semantics, our extension actually deals with two kinds of bidimensional indices, namely *variable indices* and *matchable indices*. This newly introduced calculus is simply called *Pure Pattern Calculus with de Bruijn indices* ($\mathsf{PPC_{dB}}$).

Our main result consists of showing that the relation between PPC and $\mathsf{PPC_{dB}}$ is a strong bisimulation with respect to their respective redution relations, *i.e.* they have exactly the same operational semantics. For that reason, proper translations between the two calculi were defined, $[\![\_]\!] : \mathbb{T}_{\mathsf{PPC}} \to \mathbb{T}_{\mathsf{PPC_{dB}}}$ and $(\!|\_|\!) : \mathbb{T}_{\mathsf{PPC_{dB}}} \to \mathbb{T}_{\mathsf{PPC}}$, in such a way that $[\![\_]\!]$ is the inverse of $(\!|\_|\!)$ and vice-versa (modulo α-conversion). Most notably, these embeddings are shown to preserve the matching operation of their respective domain calculus.

The strong bisimulation result allows to port into $\mathsf{PPC_{dB}}$ many already known results for PPC. Of particular interest for our line of research are the confluence and the existence of normalising reduction strategies for PPC [4], a rather complex result that requires dealing with notions of *gripping* [21] and *necessary sets* [26] of redexes. The result introduced on this paper may allow for a direct implementation of such strategies without the inconveniences of working modulo α-conversion.

As commented before, the ultimate goal of our research is the implementation of a prototype for a typed functional programming language capturing *path polymorphism*. This development is based on the *Calculus of Applicative Patterns* (CAP) [1], for which a static type system has already been introduced, guaranteeing well-behaved operational semantics, together with its corresponding efficient type-checking algorithm. CAP is essentially the static fragment of PPC, where the abstraction is generalised into an alternative (*i.e.* abstracting multiple branches at once). These two calculi are shown to be equivalent. Thus, future lines of work following the results presented in this paper involve porting CAP to the bidimensional indices setting and formalising the ideas of [4] into such framework. This will lead to a first functional version of the sought-after prototype.

References

[1] Ayala-Rincón, M., E. Bonelli, J. Edi and A. Viso, *Typed path polymorphism*, Theor. Comput. Sci. **781** (2019), pp. 111–130.
URL https://doi.org/10.1016/j.tcs.2019.02.018

[2] Barendregt, H. P., "The lambda calculus - its syntax and semantics," Studies in logic and the foundations of mathematics **103**, North-Holland, 1985.

[3] Bonelli, E., D. Kesner, C. Lombardi and A. Ríos, *Normalisation for dynamic pattern calculi*, in: A. Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, LIPIcs **15** (2012), pp. 117–132.
URL https://doi.org/10.4230/LIPIcs.RTA.2012.117

[4] Bonelli, E., D. Kesner, C. Lombardi and A. Ríos, *On abstract normalisation beyond neededness*, Theor. Comput. Sci. **672** (2017), pp. 36–63.
URL https://doi.org/10.1016/j.tcs.2017.01.025

[5] Bonelli, E., D. Kesner and A. Ríos, *de Bruijn indices for metaterms*, J. Log. Comput. **15** (2005), pp. 855–899.
URL https://doi.org/10.1093/logcom/exi051

[6] Cerrito, S. and D. Kesner, *Pattern matching as cut elimination*, Theor. Comput. Sci. **323** (2004), pp. 71–127.
URL https://doi.org/10.1016/j.tcs.2004.03.032

[7] Cirstea, H. and C. Kirchner, *ρ-calculus. Its Syntax and Basic Properties*, in: *CCL*, 1998, pp. 66–85.

[8] de Bruijn, N. G., *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem*, Indagationes Mathematicae **75** (1972), pp. 381–392.

[9] de Bruijn, N. G., "A namefree lambda calculus with facilities for internal definition of expressions and segments," EUT report. WSK, Dept. of Mathematics and Computing Science, Technische Hogeschool Eindhoven, 1978.

[10] Edi, J., A. Viso and E. Bonelli, *Efficient type checking for path polymorphism*, in: T. Uustalu, editor, *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia*, LIPIcs **69** (2015), pp. 6:1–6:23.
URL https://doi.org/10.4230/LIPIcs.TYPES.2015.6

[11] Glauert, J. R. W., D. Kesner and Z. Khasidashvili, *Expression reduction systems and extensions: An overview*, in: A. Middeldorp, V. van Oostrom, F. van Raamsdonk and R. C. de Vrijer, editors, *Processes, Terms and Cycles: Steps on the Road to Infinity, Essays Dedicated to Jan Willem Klop, on the Occasion of His 60th Birthday*, Lecture Notes in Computer Science **3838** (2005), pp. 496–553.
URL https://doi.org/10.1007/11601548_22

[12] Jay, B., *The pattern calculus*, ACM Trans. Program. Lang. Syst. **26** (2004), pp. 911–937.
URL https://doi.org/10.1145/1034774.1034775

[13] Jay, B., "Pattern Calculus - Computing with Functions and Structures," Springer, 2009.
URL https://doi.org/10.1007/978-3-540-89185-7

[14] Jay, B. and D. Kesner, *Pure pattern calculus*, in: P. Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, Lecture Notes in Computer Science **3924** (2006), pp. 100–114.
URL https://doi.org/10.1007/11693024_8

[15] Jay, B. and D. Kesner, *First-class patterns*, J. Funct. Program. **19** (2009), pp. 191–225.
URL https://doi.org/10.1017/S0956796808007144

[16] Kahl, W., *Basic pattern matching calculi: Syntax, reduction, confluence, and normalisation* (2003).

[17] Kamareddine, F. and A. Ríos, *A lambda-calculus à la de Bruijn with explicit substitutions*, in: M. V. Hermenegildo and S. D. Swierstra, editors, *Programming Languages: Implementations, Logics and Programs, 7th International Symposium, PLILP'95, Utrecht, The Netherlands, September 20-22, 1995, Proceedings*, Lecture Notes in Computer Science **982** (1995), pp. 45–62.
URL https://doi.org/10.1007/BFb0026813

[18] Klop, J. W., V. van Oostrom and R. C. de Vrijer, *Lambda calculus with patterns*, Theor. Comput. Sci. **398** (2008), pp. 16–31.

[19] Martín, A., A. Ríos and A. Viso, *Pure pattern calculus* à la *de Bruijn*, Extended report (2020), https://arxiv.org/abs/2006.07674.

[20] Mayr, R. and T. Nipkow, *Higher-order rewrite systems and their confluence*, Theor. Comput. Sci. **192** (1998), pp. 3–29.
URL https://doi.org/10.1016/S0304-3975(97)00143-6

[21] Melliès, P.-A., "Description Abstraite des Systèmes de Réécriture," Ph.D. thesis, Université Paris VII (1996).

[22] Nipkow, T., *Higher-order critical pairs*, in: *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991* (1991), pp. 342–349.
URL https://doi.org/10.1109/LICS.1991.151658

[23] van Oostrom, V., *Lambda calculus with patterns*, Technical Report IR-228, Vrije Universiteit, Amsterdam (1990).

[24] van Oostrom, V. and F. van Raamsdonk, *The dynamic pattern calculus as a higher-order pattern rewriting system*, in: *7th International Workshop on Higher-Order Rewriting, HOR 2014, Held as Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 9-24, 2014. Proceedings*, 2014.

[25] van Raamsdonk, F., "Confluence and Normalization for Higher-Order Rewriting," Ph.D. thesis, Amsterdam University (1996).

[26] van Raamsdonk, F., *Outermost-fair rewriting*, in: P. de Groote, editor, *Typed Lambda Calculi and Applications, Third International Conference on Typed Lambda Calculi and Applications, TLCA '97, Nancy, France, April 2-4, 1997, Proceedings*, Lecture Notes in Computer Science **1210** (1997), pp. 284–299.
URL https://doi.org/10.1007/3-540-62688-3_42

[27] Viso, A., "Un estudio semántico sobre extensiones avanzadas del λ-cálculo: patrones y operadores de control," Ph.D. thesis, Universidad de Buenos Aires (2020).

[28] Viso, A., E. Bonelli and M. Ayala-Rincón, *Type soundness for path polymorphism*, in: M. R. F. Benevides and R. Thiemann, editors, *Proceedings of the Tenth Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2015, Natal, Brazil, August 31 - September 1, 2015*, Electronic Notes in Theoretical Computer Science **323** (2015), pp. 235–251.
URL https://doi.org/10.1016/j.entcs.2016.06.015