The rho cube : some results, some problems

Horatiu Cirstea, Claude Kirchner, Luigi Liquori, Benjamin Wack

The Rho cube : some results, some problems

Horatiu Cirstea [1], Claude Kirchner [1], Luigi Liquori [1] and Benjamin Wack [2]

[1] *INRIA & LORIA 54506 Vandoeuvre-lès-Nancy, BP 239 Cedex FRANCE*

[2] *ENS LYON 46, allée d'Italie 69364 Lyon Cedex 07 FRANCE*

{*cirstea, ckirchne, lliquori, wack*}*@loria.fr*

Abstract

The rewriting calculus, also called ρ-calculus, embeds in a same setting the λ-calculus and the rewriting, by allowing abstraction not only on variables but also on patterns. The higher-order mechanisms of the λ-calculus and the pattern matching facilities of the rewriting are then both available at the same level. It is worth noticing that the complexity of the calculus breaks the confluence property, so that we need to define appropriate strategies, or restrictions, in order to recover it.

We choose here to look at the ρ-calculus as an extension of the λ-calculus, and we study the typed aspects. Our study is based upon a generalization of Barendregt's λ-cube, in which we unify both abstractors λ and Π into a single one. We need to deal with the original features of the ρ-calculus too: matching power, non-determinism, confluence issues.

With proper restrictions, we have proved most of the usual properties for typed calculi: substitution lemma, correctness of types, subject reduction, consistency. However, the uniqueness of typing is no longer valid mainly because of the unification of both abstractors. We are able to describe how many distinct types a term can admit and we can still prove the uniqueness of typing for the two most restrictive systems, $\rho\rightarrow$ and $\rho 2$.

Keywords: Rewriting calculus, rewriting, lambda calculus, type theory, Barendregt's cube, uniqueness of typing.

1 Introduction

1.1 Position

The ability to discriminate patterns is one of the main basic mechanisms the human reasoning is based on; as one commonly says "one picture is better than a thousand explanations". Indeed, the ability to recognize patterns,

i.e. pattern matching, is present since the beginning of information processing modeling. Instances of it can be traced back to pattern recognition and it has been extensively studied when dealing with strings [22], trees [17] or feature objects [1].

This paradigm appears as the core mechanism of the rewriting, which has many applications in computer science, for theoretical concerns as well as programming disciplines. For instance, the rewriting appears in the inference rules that describe a logic, and can be used as the basis of a theorem prover. It allows one to ensure certain properties of programs described in a language with a strong semantics, and therefore obtain proved software. At the implementation level, rewriting can constitute the core of a programming language and then be implicitly used by the programmer, like in Mathematica. Conversely, it can appear directly in the language if there are some pattern matching features, like in Prolog or ML.

The *rewriting calculus*, by unifying the λ-calculus and the rewriting, makes all the basic ingredients of rewriting explicit objects, in particular the notions of *rule application* and *result*. Pattern matching can therefore be used widely, and a rewriting rule becomes a first-class object, which can be created, manipulated and customized by the calculus itself.

It is well-known that the typed λ-calculi have a logical meaning: the Curry-Howard isomorphism establishes that a type is also a proposition, and a term is also a proof of the proposition corresponding to its type (for more details, see [27] for instance). An auxiliary result is that the expressiveness of the corresponding logic depends widely on the considered type system. Since the ρ-calculus includes rewriting and is thus more powerful than the λ-calculus, it seems natural to wonder what the consequences will be on the corresponding logics. This paper deals with the first step of this issue, namely the metatheoretical study of various type systems for the ρ-calculus.

1.2 Related works

Some other studies have been conducted about the relations between rewriting, λ-calculusand types. Th. Coquand has shown how to type some *case* expressions with dependent types in order to do safe pattern matching [12]. D. Kesner, L. Puel and V. Tannen have proposed a typed pattern calculus [20], which has been designed as a computational interpretation of the Gentzen sequent proofs for the intuitionistic propositional logic. F. Blanqui has studied type systems very similar to the *Pure Type Systems*, but with an arbitrary conversion rule; the main part of his work is dedicated to the case when this relation is given by a rewriting system [5]. We will start from the ρ-calculus, which is already provided with its operational semantics, and try to understand in which way it extends the semantics of the associated type systems.

1.3 Roadmap of the paper

In Section 2, we will describe the syntax and the evaluation rule of the ρ-calculus. We will see that the matching process can be parametrized with an arbitrary theory; the resulting possible non-determinism is handled by the calculus itself as a collection of results. The expressiveness of the calculus will be overviewed on some encoding examples.

To end with this operational description of the calculus, we will deal with confluence issues. In fact, the plain ρ-calculus is not confluent, mainly because of too powerful pattern matching features. We rely on the strategies and restrictions described in [9,3] to recover confluence.

Section 3 is straightforwardly adapted from [11]: we build nine different type systems for the ρ-calculus. The first eight are directly inspired by Barendregt's λ-cube systems [2]. The ninth one is more powerful than any of the others, since it is derived from the *Extended Calculus of Constructions* (ECC) of Z. Luo [23]. In ECC, indeed, we have an infinite set of sorts, *i.e.* $s \in \{*, \Box_i\}$, with $i \in \mathbb{N}$, and the extra axiom $\vdash \Box_i : \Box_{i+1}$ with $i \in \mathbb{N}$.

The issue of introducing reduction at the level of types has already been addressed in [6,19]: the authors proposed to add a Π-reduction rule, and to consider a more general $\beta\Pi$-conversion relation. Some cumbersome problems, such as failure of the subject reduction, arise with this setting, and require to extend the basic λ-calculus to solve them. In the ρ-calculus, we choose to make a *complete* unification of both operators "λ" and "Π" into the only abstraction symbol present in the ρ-calculus, *i.e.* the "$\rightarrow$" operator. As a result, the abstraction mechanism becomes totally uniform, and reduction can happen even in types. The system ρECC, with its infinite countable set of sorts, turns out to be crucial then, since it appears in nearly every typing property.

To go further in the study of the typed ρ-calculus, preservation of the type under reduction and confluence are avoidable. As we will see, the substitutions have to be enhanced so that they take types into account during pattern matching. Moreover, we need an unusual typing rule to deal with reductions that are blocked by the strategies for confluence.

In Section 4, we study the metatheoretical properties of these nine type systems. The lemmas and their proofs are mostly derived of the survey from Barendregt. We prove the preservation of typing judgments by generalized substitutions, the correctness of types, the subject reduction and the consistency of strongly normalizing terms. To adapt the proofs which exist for the λ-calculus, we need to fetch more information in the typing derivations, in order to deal with the new elements of our calculus. For correctness and subject reduction, we have only weak versions since the conclusions are only available in the most powerful system ρECC, whatever is the considered system in the assumptions.

In Section 5, we deal with uniqueness issues. In fact, we quickly show that the unification of both abstractors λ and Π breaks this property for most systems. We can even build terms with an arbitrary number of distinct types; still, a given term has always a finite number of types. We end up with the proof that uniqueness remains valid in the polymorphic system $\rho 2$, which is more or less an extended model of ML; this results holds also for the simply typed calculus $\rho\!\rightarrow$, since it is a subsystem of $\rho 2$.

2 The rewriting calculus

This section gives the basis of the ρ-calculus: its syntax, its semantics and some examples about expressiveness. The typed aspects are left apart for the moment ; we give some hints about the failure of confluence and the ways to recover it.

2.1 Operational definition

2.1.1 Static syntax

Let us begin with some notation conventions: the capitals A, B, C, D... generally refer to a ρ-term. A sort can be noted s, s', s_1, s_2, etc. Variables are generally taken into X, Y, Z, X_1, X_2... Constants will be given by minuscule letters : a, b, c... for symbols without an argument, and f, g, h for functions. When we want to talk about a term which is either a variable or a constant, we will use the greek letters α, β, etc. The syntactic equivalence of two terms is noted $\equiv$.

The formal syntax of the ρ-calculus is the following:

Sorts	$\mathcal{S} ::= * \mid \Box \mid \Box_1 \mid \cdots \mid \Box_n \mid \ldots$
Variables	$\mathcal{V}ar ::= X, Y, Z, \ldots, X_1, X_2, \ldots, X_n \ldots$
Constants	$\mathcal{C}st ::= a, b, c, \ldots, f, g, h \ldots$
Terms	$\mathcal{T} ::= \mathcal{S} \mid \mathcal{V}ar^{\mathcal{T}} \mid \mathcal{C}st^{\mathcal{T}} \mid \mathcal{T} \rightarrow \mathcal{T} \mid \mathcal{T}\bullet\mathcal{T} \mid null \mid \mathcal{T}, \mathcal{T}$

Let us explain these different elements:

Sorts have the same signification than in the typed λ-calculus and in the *Pure Type Systems*: they are "types of types".

Variables are given as a countable set of identifiers. Each variable X is given with its type A, with the notation X^A. It is somewhat the same spirit as in the typing *à la Church*: we try to include in the term itself all the information needed to typecheck it. Contexts are then no longer necessary, as long as we respect a coherence constraint (def. 2.1).

Constants are atoms that can never be instantiated. They are decorated with their types too, with the same condition 2.1 as for variables. The application of a given "function symbol" f to some terms $t_1, \ldots, t_n$ will be

denoted by the application operator $\bullet$: it is typically a currified notation $(((f \bullet t_1) \bullet \ldots) \bullet t_n$, so we do not define a fixed arity for constants.

Abstraction is denoted by $\rightarrow$, which has been taken from the rewriting community. The term $A \rightarrow B$ corresponds roughly to the same rewriting rule, but it can be the result or the argument of another rule. Relatively to the λ-calculus, the signification of the abstractors $\rightarrow$ and λ is the same, but $\rightarrow$ is much more general since abstraction can be done on patterns, not only on variables. In a typed context, $\rightarrow$ will represent both abstractors λ and Π.

Application is denoted by $\bullet$. We choose to explicit an operator which has no symbol in the λ-calculus because we will use it more generally (for functional terms as well as for function symbols).

Structures are used to deal with non-determinism. The empty structure *null* represents an application failure; a singleton is a plain term and it represents a deterministic result; a collection of results $t_1, \ldots, t_n$ represents a non-deterministic choice between the elements of the collection. We must stress that the operator "," is part of the calculus itself, and that it can be customized with some theories to get a particular structure such as a list, a set or a multiset.

Definition 2.1 (Tower convention)
If the same variable or constant α appears twice in the same term (or in the same type inference) with the decorations α^A and α^B, the $A \equiv B$. This corresponds to correctness of the context in the typed λ-calculus.

If there is no ambiguity, we will consider as much as possible that $\bullet$ is left associative and $\rightarrow$ is right associative (like in the λ-calculus). We will omit useless braces with the convention that $\bullet$ is prior to $\rightarrow$ which is in turn prior to ",".

To connect completely the ρ-calculus with the λ-calculus, we have to extend the two following notions:

Free and bound variables : an abstraction $A \rightarrow B$ is more general than a $\lambda x.B$ since its "scope" is the whole pattern A. All the variables in A must then be bound, except for the variables in the typing decorations (in λ-calculus, they would either be in the context and therefore would be free, or bounded by an other λ). Hence, we need to distinguish the "ground variables" $FV^{\downarrow}$ in the term from the variables of the typing decorations $FV^{\uparrow}$.

Definition 2.2 (Free and bound variables for a ρ-term)

$$\text{(i)}\quad FV^{\downarrow}(a^A) \triangleq \emptyset \qquad FV^{\downarrow}(X^A) \triangleq \{X\}$$
$$FV^{\downarrow}(A \bullet B) \triangleq FV^{\downarrow}(A,B) \triangleq FV^{\downarrow}(A) \cup FV^{\downarrow}(B)$$
$$FV^{\downarrow}(A \to B) \triangleq FV^{\downarrow}(B) \setminus FV^{\downarrow}(A)$$
$$\text{(ii)}\quad FV^{\uparrow}(a^A) \triangleq FV^{\uparrow}(X^A) \triangleq FV(A)$$
$$FV^{\uparrow}(A \bullet B) \triangleq FV^{\uparrow}(A,B) \triangleq FV^{\uparrow}(A) \cup FV^{\uparrow}(B)$$
$$FV^{\uparrow}(A \to B) \triangleq (FV^{\uparrow}(A) \cup FV^{\uparrow}(B)) \setminus FV^{\downarrow}(A)$$
$$\text{(iii)}\quad FV(null) \triangleq FV(s) \triangleq \emptyset$$
$$FV(A) \triangleq FV^{\downarrow}(A) \cup FV^{\uparrow}(A)$$
$$\text{(iv)}\quad BV(A) \triangleq \mathcal{V}ar(A) \setminus FV(A)$$

All along this paper, we work modulo α-conversion and Barendregt's hygiene convention; this ensures that free and bound variables have distinct names and that no capture occurs during reductions. To legitimate this, let us note that H. Cirstea has studied a ρ-calculus with explicit substitutions in [8], which solves the problem of variables renaming.

Substitutions : With these conventions, we can apply substitutions quite straightforwardly, without forgetting to deal with typing decorations:

$$\sigma\ s \triangleq s \qquad \sigma\ a^A \triangleq a^{\sigma A}$$

$$\sigma(A \diamond B) \triangleq \sigma A \diamond \sigma B \text{ (where } \diamond \in \{\to, \bullet, ,\})$$

$$\sigma\ Y^A \triangleq \begin{cases} Y^{\sigma A} & \text{if } Y \notin \mathcal{D}om\ \sigma \\ B & \text{if } \sigma = [\ldots Y/B \ldots] \end{cases}$$

2.1.2 Evaluation of a ρ-term

The core mechanism of the ρ-calculus is pattern matching. To do it effectively, we need to choose an equality for comparison of two terms. In the ρ-calculus, this is done very generally by allowing the "user" to choose an arbitrary theory $\mathbb{T}$ which defines all the equality axioms. We only assume that the equality $\overset{\mathbb{T}}{=}$ is a congruence, by requiring that it verifies reflexivity, symmetry, transitivity and stability by context.

In practice, we will use quite simple theories :

- **The empty theory** $\mathbb{T}_\emptyset$ is just the syntactical equivalence $\equiv$ on terms. It has the interesting property of generating a deterministic pattern matching.
- **The theory of commutativity** $\mathbb{T}_c^f$ adds the rule $\Vdash f\ A\ B \overset{\mathbb{T}_c^f}{=} f\ B\ A$
- **The theory of associativity** $\mathbb{T}_a^f$ adds the rule $\Vdash f(f\ A\ B)\ C \overset{\mathbb{T}_a^f}{=} f\ A\ (f\ B\ C)$
- **The theory of idempotency** $\mathbb{T}_i^f$ adds the rule $\Vdash f\ A\ A \overset{\mathbb{T}_i^f}{=} A$

- **More subtle theories** can even lead to infinite classes of equivalence, and thus possibly infinite results in pattern matching.

An interesting point is that the symbol f in these examples can be any constant of the ρ-calculus, but it can even be the operator ",". This allows one to customize the structure of a non-deterministic result: for instance, a commutative coma generates a an ordered list; if it is associative too, it is a multiset; finally, with idempotency we get a set.

The pattern matching process can then be formalized properly:

Definition 2.3 (Matching equation and solution)
Given an equational theory $\mathbb{T}$ *on terms:*

(i) *a* matching equation *is denoted by* $A \ll_{\mathbb{T}} B$ *;*

(ii) *a substitution* σ *is a* solution *of* $A \ll_{\mathbb{T}} B$ *if* $\sigma A \stackrel{\mathbb{T}}{=} B$.

For commodity of notations, we will define a function $\mathcal{S}ol$ from matching equations to substitutions:

(i) If a matching equation S admits any substitution σ as a solution, it is said to be *trivial*. In this particular case, we consider that the only relevant substitution is the identity $\mathbb{ID}$: $\mathcal{S}ol(\mathsf{S}) \triangleq \{\mathbb{ID}\}$.

(ii) If S admits no solution, $\mathcal{S}ol(\mathsf{S}) \triangleq \emptyset$.

(iii) Elsewise, $\mathcal{S}ol(\mathsf{S}) \triangleq \{\sigma \mid \sigma \text{ is a solution of } \mathsf{S}\}$.

It is quite well-known [18] that for the syntactic matching, *i.e.* with the empty theory $\mathbb{T}_\emptyset$, $\mathcal{S}ol(\mathsf{S})$ will always be either empty or a singleton, and that it can be computed in linear time of the size of the matching equation S.

By now we have settled all the background necessary to describe the reduction rules of the rewriting calculus:

$$
\begin{array}{rcl}
(A \to B)\bullet C & \mapsto_\rho & \begin{cases} \sigma_1 B, \ldots, \sigma_n B, \ldots & \text{where } \mathcal{S}ol(A \ll_{\mathbb{T}} C) = \{\sigma_1, \ldots, \sigma_n \ldots\} \\ null & \text{if } \quad \mathcal{S}ol(A \ll_{\mathbb{T}} C) = \emptyset \end{cases} \\
(A, B)\bullet C & \mapsto_\delta & A\bullet C, B\bullet C \\
null\bullet C & \mapsto_\nu & null
\end{array}
$$

Fig. 1. The reduction rules for the ρ-calculus

Let us quickly explain their meaning:

ρ-reductions : the "rewriting" part of the calculus appears here, since pattern matching is used in the computation of $\mathcal{S}ol$. We see here how the choice of the theory can have great consequences on the evaluation: there can be an arbitrarily high number of solutions to the matching equation, generating as many terms in the structure of results. When $\mathcal{S}ol$ is empty, a *matching*

failure has occured; it is the only way of producing *null* without having it before as a subterm.

δ-reduction : this rule distributes structures on the left hand side of the operator $\bullet$. It allows, for instance, to apply in parallel two distinct rewriting rules A and B to the same term C.

ν-reduction : in fact it is just the converse case of δ-reduction, with an empty structure before $\bullet$.

In fact, these are the top-level rules. Like in most calculi, we have the following congruence condition:

$$\text{If } A \mapsto_{\rho\delta\nu} A', \text{ then } \mathsf{Ctx}[A] \mapsto_{\rho\delta\nu} \mathsf{Ctx}[A']$$

2.2 Examples (expressiveness matters)

To show better the expressiveness of the ρ-calculus, we informally compare it with two other formalisms: the λ-calculus, and first order rewriting systems. For readability, type decorations of variables and constants are omitted since they don't modify reductions.

- The encoding of the λ-calculus is quite straightforward: we restrict the patterns, *i.e.* the left hand sides of $\to$ to be only a single variable, the theory is empty and we drop the structures. There is then the following correspondence:

$$\begin{array}{lllll} & \lambda x.M & \cong & X \to M & \\ (\lambda x.M)N \mapsto_\beta M[N/x] & \cong & (X \to M)\bullet N & \mapsto_\rho & \left\{\sigma M \mid \sigma X \stackrel{\emptyset}{=} N\right\} \\ & & & \equiv & M[N/X] \end{array}$$

  It is easy to check that $\sigma = [N/X]$ is always the unique solution of the particular matching equation $X \ll_\emptyset N$, and that there can be no matching failure. Therefore, no structure other than a singleton can appear during reductions if it was not present before in the term translated from the λ-calculus. This translation is operationally conservative, in the sense that β-reductions are exactly mimicked by the ρ-reductions (and that δ and ν reductions can not take place).

  But constants and pattern matching of course add a computational power which is not present in the λ-calculus:

$$\begin{array}{lll} (f\bullet X \to g\bullet X)\bullet(f\bullet a) \mapsto_\rho g\bullet a & & \textit{(Extraction and re-use of a subterm)} \\ (h\bullet X\bullet X \to X)\bullet(h\bullet b\bullet b) \mapsto_\rho b & & \textit{(Comparison of two subterms)} \end{array}$$

- To understand the difference between the ρ-calculus and the rewriting systems, one has to see that a ρ-term is consumed by a ρ-reduction, and therefore can operate only locally, whereas the rules of a rewriting system are

applied as many times as possible. For instance, the simple rewriting system $\{a \to f(a)\}$ applied to a is obviously non terminating because it reduces infinitely to terms of the shape $f^n(a)$. In the ρ-calculus, we can have a control over the application of this rule :

$$\begin{aligned} (a \to f\bullet a)\bullet a &\mapsto_\rho f\bullet a \text{ (which is in normal form)} \\ (f\bullet a \to f\bullet(f\bullet a))\bullet((a \to f\bullet a)\bullet a) &\mapsto_\rho (f\bullet a \to f\bullet(f\bullet a))\bullet(f\bullet a) \\ &\mapsto_\rho f\bullet(f\bullet a) \text{ (normal form again)} \end{aligned}$$

It can be shown that, for every rewriting system $\mathcal{R}$ and every reduction path $\mathcal{P} = t \mapsto_R t'$ beginning with a certain term t, one can build a ρ-term $\xi_{\mathcal{R},t,\mathcal{P}}$ such that $\xi_{\mathcal{R},t,\mathcal{P}}\bullet t \mapsto_\rho^* t'$. If we extend the language with a term that tests failure of the application of a rule, it is possible to define a ρ-term which simulate the *global* behaviour of a rewriting system: $\xi_{\mathcal{R}}$ will depend only on $\mathcal{R}$, and for any term t such that $t \mapsto_R t'$, $\xi_{\mathcal{R}}\bullet t \mapsto_\rho^* t'$. We can even choose a certain strategy in the application of the rules, for example the innermost application.

To end with the rewriting, let us show a practical application of the choice of the theory. Here we use an (infix) operator $\oplus$ in the signature, which we can choose to be either syntactic, associative, commutative or associative-commutative :

$$\begin{aligned} (X \oplus Y \to X)\bullet(a \oplus (b \oplus c)) &\mapsto_\emptyset a \\ (X \oplus Y \to X)\bullet(a \oplus (b \oplus c)) &\mapsto_A a, a \oplus b \\ (X \oplus Y \to X)\bullet(a \oplus (b \oplus c)) &\mapsto_C a, b \oplus c \\ (X \oplus Y \to X)\bullet(a \oplus (b \oplus c)) &\mapsto_{AC} a, b, c, a \oplus b, b \oplus c, a \oplus c \end{aligned}$$

- The encoding of various other formalisms in the ρ-calculus has been studied: the Object Calculus $\lambda\mathcal{O}bj$ of Abadi and Cardelli can be simply encoded if "," is associative and idempotent (so that the structures are lists) [10]; Combinatory Reduction Systems [21], which inject higher-order mechanisms in the rewriting, can be encoded with an appropriate theory $\mathbb{T}_{\mathtt{P}}$ which ensures that patterns are in the required shape [4].

3 Confluence issues

3.1 Examples of non-confluent reductions

The non-confluence of the plain ρ-calculus is easily seen on counterexamples:

The first reason of non-confluence (examples of fig. 2) is the possibility of two distinct reduction paths, one leading to a value, the other to *null*. As shown in fig. 3, it would not be sufficient to eliminate matching failure, since a pattern with the shape $X\bullet\ldots$ (where X is said to be *active*) can lead to

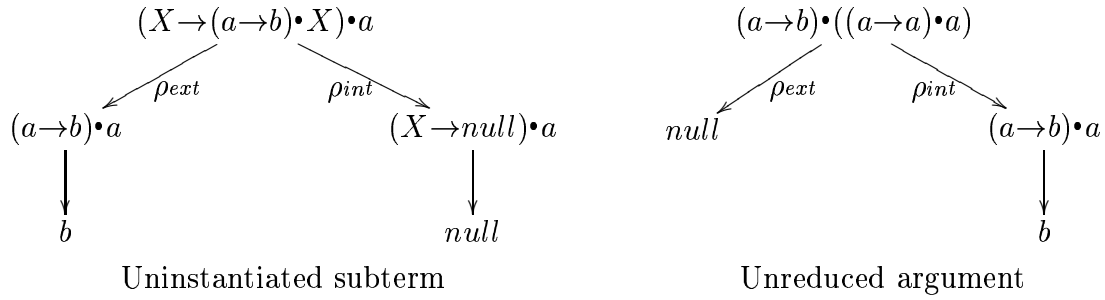

Fig. 2. Non-confluence by failure

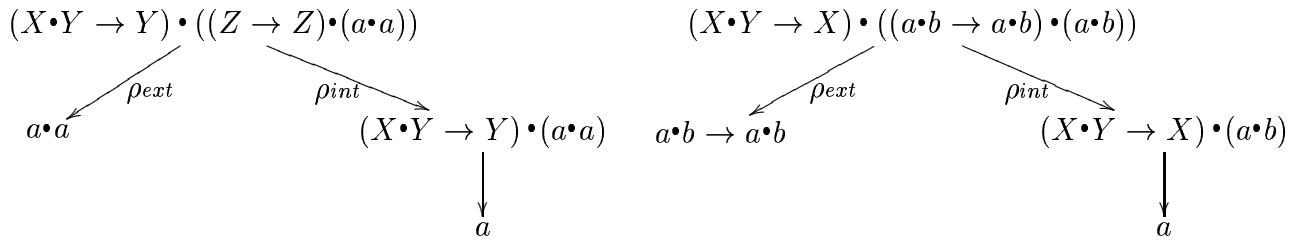

Fig. 3. Non-confluence by active variable

two distinct values. One can check that these are the two main reasons for non-confluence.

The non-confluence is a problem for programming, since we are interested in deterministic results, but also for typechecking, since the conversion rule introduces reduction in the typing process.

3.2 Recovering confluence

A first idea is to consider the ρ-calculus as the model of a programming language, and to build strategies that always yield the same result when two distinct reductions would be possible. H. Cirstea has proposed two strategies [9], mainly inspired by G. Plotkin's call-by-value [26]:

- The ρ-calculus with values ρ_v :
  We define *values* as terms with the restriction that application have the shape $\alpha^{\mathcal{T}} \bullet \mathcal{V} \bullet \ldots \bullet \mathcal{V}$
  $(A \to B) \bullet C$ is reduced only if A and C are values and C is closed. The conditions on C ensure that there will be no ambiguity between two reductions since the argument is a value, which is more or less a normal form.
  Reductions $\mapsto_\delta$ and $\mapsto_\nu$ are left as is.
- The ρ-calculus with rigid values ρ_{rv}^{ok} :

We define *rigid values* in the same fashion, but without active variables: applications are $a^{\mathcal{T}}\bullet\mathcal{RV}\bullet\ldots\bullet\mathcal{RV}$

$(A \to B)\bullet C$ is reduced only if A is a rigid value and there is no matching failure. The condition on A implies that there is no active variable in the pattern, and the removal of matching failures prevents the first counterexamples from occuring.

Reductions $\mapsto_\delta$ are left as is, and $\mapsto_\nu$ disappears since there is no more *null* (no matching failures).

The proof of confluence is directly inspired by Tait & Martin-Löf's classical proof. In the study of the type systems, we will assume more generally that we use such a strategy, without caring about which one it is; the only assumption we need is that the reducibility of a redex $(A \to B)\bullet C$ only depends on the pattern A and on the argument C.

Another way of recovering confluence is to restrict the basic syntax of the calculus itself, and then prove that it is not broken by reductions. L. Liquori has shown [3] that V. van Oostrom's *Rigid Pattern Condition* [28] remains valid for the ρ-calculus:

Definition 3.1 (Rigid Pattern Condition)
A pattern $\mathsf{Ctx}[X_1, \ldots X_n]$ *is rigid if its shape* $\mathsf{Ctx}[]$ *is preserved by substitution of the* X_i *and reduction.*

In particular, if a term is linear, in normal form and without active variables, it is a rigid pattern.

Then, if we allow only abstractions with shape $\mathcal{P} \to \mathcal{T}$ where $\mathcal{P}$ is a rigid pattern, the calculus is confluent. In the study of the type system, this restriction can be used instead of the strategies since it concerns only the patterns. It has the further interest that it is only a syntactical restriction; hence, it seems more appropriate if we consider the ρ-terms as proof terms and want to do cut elimination.

4 A Barendregt-like type system

4.1 The ρ-cube

As hinted by their name, our type systems are directly inspired by Barendregt's λ-cube. This presentation is mostly adapted from [11]. The absence of context can be surprising, but all the information we need is contained in the type decorations of the variables and the constants.

4.1.1 The typing rules

Most of these rules are a mere translation of the λ-cube ones. Let us discuss them:

System	Associated rules ($\forall i, j \in \mathbb{N},\ i \leq j$)
$\rho\!\rightarrow$	$(*,*)$
$\rho 2$	$(*,*)\ (\Box_0,*)$
ρP	$(*,*)\quad (*,\Box_0)$
$\rho P2$	$(*,*)\ (\Box_0,*)\ (*,\Box_0)$
$\rho\underline{\omega}$	$(*,*)\quad (\Box_0,\Box_0)$
$\rho\omega$	$(*,*)\ (\Box_0,*)\quad (\Box_0,\Box_0)$
$\rho P\underline{\omega}$	$(*,*)\quad (*,\Box_0)\quad (\Box_0,\Box_0)$
ρCC	$(*,*)\ (\Box_0,*)\ (*,\Box_0)\quad (\Box_0,\Box_0)$
ρECC	$(*,*)\ (\Box_i,*)\ (*,\Box_i)\quad (\Box_i,\Box_j)$

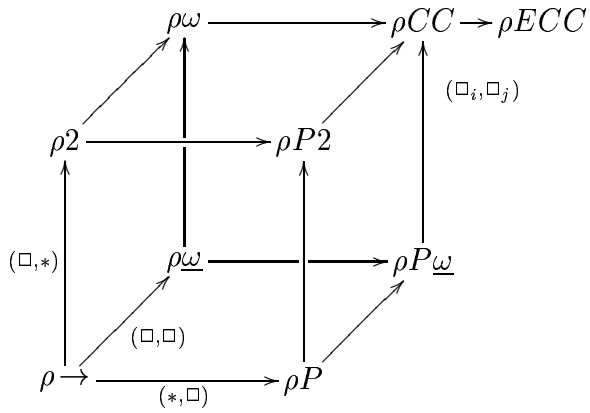

Fig. 4. The 9 (8 + 1) systems of the ρ-cube

Rules derived from the λ-cube :

$$\frac{}{\vdash * : \Box_0}\ (Axiom_*) \qquad \frac{i \in \mathbb{N}}{\vdash \Box_i : \Box_{i+1}}\ (Axiom_{\Box_i})$$

$$\frac{\vdash A : s}{\vdash \alpha^A : A}\ (Start) \qquad \frac{\vdash A : C \quad \vdash B : s \quad B =_\rho C}{\vdash A : B}\ (Conv)$$

$$\frac{\vdash B : C \quad \vdash A \to C : s}{\vdash A{\to}B : A \to C}\ (Abs) \qquad \frac{\vdash A : A' \quad \vdash A' : s_1 \quad \vdash B : s_2 \quad (s_1, s_2) \in \mathcal{R}}{\vdash A{\to}B : s_2}\ (Prod)$$

$$\frac{\vdash A : C \to D \quad \vdash C : E \quad \vdash B : E}{\vdash A{\bullet}B : (C \to D){\bullet}B}\ (Appl) \qquad \frac{\vdash A : C \to s \quad \vdash C : E \quad \vdash B : E}{\vdash A{\bullet}B : s}\ (ApplSort)$$

Structures-specific rules :

$$\frac{\vdash A : s}{\vdash null : A}\ (Null) \qquad \frac{\vdash A : C \quad \vdash B : C}{\vdash A, B : C}\ (Struct)$$

Fig. 5. Typing inference rules in the ρ-cube

$(Axiom)$: These rules describe the infinite hierarchy of universes. In the basic cube, only $(Axiom_*)$ is used, and the set of $(Axiom_{\Box_i})$ gives its power to ρECC.

$(Start)$: This rule checks the correctness of the type decorations for variables and constants. The coherence of these types all along the type inference is ensured by the tower condition 2.1.

$(Conv)$: It is exactly the same conversion rule as in the λ-calculus, but considering ρ-reduction instead of β-reduction.

(Abs) : An important innovation appears here: we *completely* unify both abstractors λ and Π in $\rightarrow$, introducing a notion of reduction at the level of types. In the type $A \rightarrow C$, we keep the same pattern A in order to have dependent types, just like when a λ-term $\lambda(x : A).b$ is typed by $\Pi(x : A).B$, with the same variable $x : A$. We can notice that an abstraction $A \rightarrow B$ could be typed either by the (Abs) rule or by the $(Prod)$ rule, which prevents one to do syntax-guided typing.

$(Prod)$: This rule derives straightforwardly from the λ-cube, except that we need to infer the type A' of the pattern, whereas in the λ-calculus the type of the variable is given in the Π-abstraction. The set $\mathcal{R}$ in which pairs (s_1, s_2) are chosen is described in the table of fig. 4 and corresponds to different logical systems:

- $(*, *)$ corresponds to the simply typed λ-calculus, and is present in all the type systems.
- $(\Box, *)$ allows to build terms depending on types, making polymorphism available. In λ-calculus, it is equivalent to J.-Y. Girard's system F [15].
- $(\Box, \Box)$ allows to build types depending on types.
- $(*, \Box)$ allows to build types depending on terms. It is used to represent sets or predicates, and corresponds in λ-calculus to the Logical Framework (LF) of Harper, Honsell and Plotkin [16].
- $(\Box, *) + (\Box, \Box)$: this combination of rules gives higher-order, and thus corresponds to Girard's system $\mathsf{F}\omega$
- $(\Box, *) + (\Box, \Box) + (*, \Box)$: the whole set of basic rules corresponds to Th. Coquand's Calculus of Constructions [13].
- $(\Box_i, \Box_j)$: by adding these rules to the former three, we get a system inspired by Z. Luo's *Extended Calculus of Constructions* [23]. The infinite hierarchy of sorts turns out to be very useful when we use $\rightarrow$ as an abstractor for types and want to typecheck it as we would do for any term. The condition $i \leq j$ arises from the fact that the rule $(*, \Box)$ already exists and that two distinct predicative universes would recreate Girard's λU paradox [14].

$(Appl)$: In the λ-calculus, the corresponding rule defines the type of $A \bullet B$ as $[B/C]D$. Here, we can benefit of the reductions at the type level to write this substitution as a redex $(C \rightarrow D) \bullet B$ which will be evaluated in the conversion rule. Therefore we have got rid of the meta-level substitution that was present in the typing inference. As in the $(Prod)$ rule, we need

to type the pattern C whereas in the λ-calculus, the abstracted variable would have been given with its type.

$(ApplSort)$: Here we reintroduce some of the "cut" that was done in the application typing rule for the λ-calculus. The reason for introducing this rule is that, with certain restrictions or strategies for confluence, the type $(C \to s)\bullet B$ given by the $(Appl)$ rule would not be reducible. We choose to do as if $C \ll_{\mathbb{T}} B$ would not lead to a matching failure; then the result of the associated ρ-reduction is some σs, but we have defined $\sigma s \stackrel{\triangle}{=} s$, so we do not even need to compute σ. This allows us to save the progress made relatively to the λ-calculus: there is still no meta-level substitution.

One can check that this rule does not add meaningless typing judgments to the type system. The worst that can happen is that $\mathcal{S}ol(C \ll_{\mathbb{T}} B) = \emptyset$; but in this case the term $A\bullet B$ is likely to reduce to $null$. Since the typing rule $(Null)$ states that any correct type can be given to $null$, it is harmless to give type s to a term that is convertible to $null$.

Remark 4.1 *Again, two distinct typing rules can be used for the same term $A\bullet B$, but this seems to be less relevant that for (Abs) and $(Prod)$. When there is no matching problem in the type $(C \to D)\bullet D$, both rules $(Appl)$ and $(ApplSort)$ are equivalent; whenever there is a problem (matching failure or reduction blocked by the strategy), the use of $(Appl)$ will seldom be useful, because we would like to have mere sorts wherever we can have them (for correctness of types).*

$(Null)$: The empty structure must admit any well-formed type. In fact, a term A is equivalent to the structure $A, null$ (with $null$ being a neutral element for ","). Following the $(Struct)$ typing rule, if $\vdash A : B$ then this structure must get type B and therefore $null$ must get type B too. As we do not know exactly what are the inhabited types, we allow $null$ to be typed by any well-formed type.

$(Struct)$: We choose to impose that all the elements of a structure have the same type. Some other options have been considered: for instance, $\vdash A, B : C, D$ if $\vdash A : C$ and $\vdash B : D$; in this case we would have to extend the $(Appl)$ rule so that it can type correctly ν-redexes.

To train the reader with these systems, let us give some examples:

- New base types can be introduced as constants of type $*$. If we want to type a structure, the derivation will look like that:

$$\dfrac{\dfrac{\vdash int^* : *}{\vdash 3^{int^*} : int^*}\ (Start) \qquad \vdash 4^{int^*} : int^*}{\vdash 3^{int^*}, 4^{int^*} : int^*}\ (Struct)$$

- As our (*Appl*) rule creates a redex in the type, it is often immediately followed by a conversion rule:

$$\dfrac{\dfrac{\vdash X^{int^*} \to X^{int^*} : X^{int^*} \to int^* \quad \vdash X^{int^*} : int^* \quad \vdash 3^{int^*} : int^*}{\vdash (X^{int^*} \to X^{int^*})\bullet 3^{int^*} : (X^{int^*} \to int^*)\bullet 3^{int^*}}\ (Appl) \qquad (X^{int^*} \to int^*)\bullet 3^{int^*} =_\rho int^*}{\vdash (X^{int^*} \to X^{int^*})\bullet 3^{int^*} : int^*}\ (Conv)$$

- Of course, all the usual terms and types from the λ-calculus can be reproduced:

$$\begin{array}{lll} \text{in } \rho 2, & \vdash X^* \to X^* : * & (X^* \to X^* \ \cong\ \lambda(\alpha : *).\alpha \ \triangleq\ \bot) \\ & \vdash X^* \to (Y^{X^*} \to Y^{X^*}) : X^* \to (Y^{X^*} \to X^*) & \\ \text{in } \rho\underline{\omega}, & \vdash X^* \to * : \Box & \\ \text{in } \rho P, & \vdash X^{int^*} \to * : \Box & \\ \text{in } \rho P\underline{\omega}, & \vdash X^* \to Y^{X^*} \to * : \Box & \\ \text{in } \rho CC, & \vdash X^* \to f^{Y^* \to Y^*} \to * : \Box & \end{array}$$

- The (*Appl*) rule is effective for the typing of terms with function symbols too:

$$\vdash f^{X^{int^*} \to int^*}\bullet 3^{int^*} : (X^{int^*} \to int^*)\bullet 3^{int^*} \ =_\rho \ int^*$$

- The ninth system ρECCallows, for example, to decorate variables with sorts other than $*$:

$$\dfrac{\dfrac{\vdash \Box : \Box_1}{\vdash X^\Box : \Box}\ (Start) \qquad \vdash \Box : \Box_1 \quad \vdash X^\Box : \Box}{\vdash X^\Box \to X^\Box : \Box}\ (Prod\ \Box_1, \Box)$$

4.2 Upgrading substitutions with types

To ensure that reductions well-behave relatively to types, we must enhance the notion of substitution we use. In the λ-calculus, an abstracted variable is given with its type, and the typing rule for applications ensures that this type is the same for the argument. In the ρ-calculus, the (*Appl*) rule checks if both the pattern C and the argument C have the same type E, but this is not sufficient to enforce that the types of all the variables in the pattern C will be preserved when applying the substitutions in $\mathcal{S}ol(C \ll_{\mathbb{T}} B)$. Therefore we need the following condition:

Definition 4.2 (Well-typed substitution)
For a given typing judgment $\vdash$ *on terms, the substitution* $\sigma = [X_1^{A_1}/B_1 \ldots X_n^{A_n}/B_n]$ *is* well-typed *if* $\quad \forall i, \ \vdash B_i : \sigma A_i$

From now, we will consider that a substitution can be accepted as a solution of a matching equation only if it is well-typed. Thus, pattern matching now depends on the theory *and* on the typing judgment. Hence, we have extended the power of the calculus since the pattern matching now has some typechecking abilities too.

Remark 4.3 *Since we add a constraint on substitutions, pattern matching will return at most as many solutions as before. Noticeably, syntactic matching will still return either a unique result or a failure.*

In all the remaining of the paper, when a substitution appears it is always the result of a pattern matching that occured during a ρ-reduction. We will then consider that all substitutions are well-typed. We will also assume that, in the strategy used for confluence, if $\sigma \in \mathcal{S}ol(A \ll_{\mathbb{T}} B)$, then a redex looking like $(X \to C)\bullet\sigma X$ can be reduced (with $X \in \mathcal{D}om\ \sigma$). One can check that it is the case for both strategies described in the previous section; it is true too for any strategy that preserves the ability of encoding the λ-calculus (because in $(X \to C)\bullet\sigma X$ would correspond to $(\lambda x.C)\sigma X$).

5 Properties of the type systems

All those properties are adapted from the classical properties of the typed λ-calculus. We suppose the calculus to be restricted enough so that it is confluent, otherwise the conversion rule would be meaningless. If no particular system is given, the typing judgment $\vdash$ of the conclusion is in the same system as the assumptions. All the lemmas of this section are true for any of the nine systems of the ρ-cube.

The two first lemmas are mostly technical and will be used in the subsequent proofs. They show that substitutions comply with conversion and typing. The condition about the domain $\mathcal{D}om$ and the codomain $\mathcal{R}an$ will always be true when σ is the solution of a matching equation, because of the hygiene convention on bound and free variables.

Lemma 5.1 (Stability of convertibility up to substitution)
For any terms A, B, for any substitution σ such that $\mathcal{D}om\ \sigma \cap \mathcal{R}an\ \sigma = \emptyset$,

$$A =_\rho B \ \Rightarrow\ \sigma A =_\rho \sigma B$$

Proof : We encode σ as a set of reductions to include it in the conversion:

$$\begin{array}{lll}
\sigma A \equiv [X_1/\sigma X_1\ X_n/\sigma X_n]A & & \text{supposing } \mathcal{D}om\ \sigma = \{X_i\} \\
\quad \equiv [X_n/\sigma X_n]...[X_1/\sigma X_1]A & & \text{as } \mathcal{D}om\ \sigma \cap \mathcal{R}an\ \sigma = \emptyset, \\
& & \text{the order of the } X_i \text{ is not relevant} \\
\quad =_\rho (X_1 \to \ldots X_n \to A)\bullet\sigma X_1\bullet\ldots\bullet\sigma X_n & & \text{because } \mathcal{S}ol(X_i \ll_{\mathbb{T}} \sigma X_i) = \{[X_i/\sigma X_i]\} \\
\quad =_\rho (X_1 \to \ldots X_n \to B)\bullet\sigma X_1\bullet\ldots\bullet\sigma X_n & & \text{because } A =_\rho B \text{ and } \mapsto\rho \text{ is congruent} \\
\quad =_\rho \sigma B & & \text{by reversing the encoding of } \sigma
\end{array}$$

□

Lemma 5.2 (Stability of the typing judgments up to substitution)

$$\forall\ \sigma, B, C \text{ such that } \vdash B : C$$
$$\text{and } \sigma \text{ is a well-typed substitution such that } \mathcal{D}om\ \sigma \cap \mathcal{R}an\ \sigma = \emptyset$$
$$\text{We have } \vdash \sigma B : \sigma C$$

Proof : by induction on the derivation of $\vdash B : C$; we distinguish cases for the last applied rule:

($Axiom$) $B \equiv s$ and $C \equiv s'$, so $\sigma B \equiv B$ and $\sigma C \equiv C$, and immediately $\vdash \sigma B : \sigma C$.

($Start$) $B \equiv \alpha^C$ and $\vdash C : s$. If $B \in \mathcal{D}om\ \sigma$, by well-typedness of the substitution we know that $\vdash \sigma(\alpha^C) : \sigma C$.

If $\alpha \notin \mathcal{D}om\ \sigma$ (or α is a constant), the induction hypothesis gives $\vdash \sigma C : \sigma s = s$, so $\vdash \sigma B = \sigma\alpha^C = \alpha^{\sigma C} : \sigma C$.

($null$) $B \equiv null$ and $\vdash C : s$; by induction hypothesis, $\vdash \sigma C : \sigma s = s$, thus $\vdash \sigma B = null : \sigma C$

($Struct, Abs, Appl, ApplSort, Prod$) All these rules are treated the same way, by applying induction hypothesis and the notion of substitution given in subsection 2.1.1.

($Conv$) Here the assumptions are that $\vdash B : C'$, $\vdash C : s$ and $C =_\rho C'$; the induction hypothesis ensures that $\vdash \sigma B : \sigma C'$ and $\vdash \sigma C : \sigma s = s$. The previous lemma 5.1 then ensures that $\sigma C =_\rho \sigma C'$, so we can apply the conversion rule again to derive $\vdash \sigma B : \sigma C$.

□

The generation lemma gives syntax-guided information about the type of a given term. It is more ambiguous than in the λ-calculus since abstractions and applications can be typed by two distinct rules.

Lemma 5.3 (Generation in the ρ-cube)
Given a typable ρ-term A (meaning that there is a B such that $\vdash A : B$), then:

(i) *if* $A \equiv s$, *then* $\exists s',\ B =_\rho s'$ *and* $\vdash s : s'$

(ii) *if* $A \equiv \alpha^C$, *then* $B =_\rho C$

(iii) *if* $A \equiv A_1, A_2$, *then* $\vdash A_1 : B$ *and* $\vdash A_2 : B$

(iv) *if* $A \equiv A_1 \to A_2$, *then :*

- *either* $\exists s, C$ *with* $\vdash A_2 : C,\ \vdash A_1 \to C : s$ *and* $B =_\rho A_1 \to C$;
- *or* $\exists s_1, s_2, C$ *with* $\vdash A_2 : s_2,\ \vdash A_1 : C : s_1$ *and* $B =_\rho s_2$

(v) *if* $A \equiv A_1 \bullet A_2$, *then* $\exists C, E$ *such that* $\vdash C : E,\ \vdash A_2 : E$ *and:*

- *either* $\exists D$ *s. t.* $\vdash A_1 : C \to D$ *and* $B =_\rho (C \to D) \bullet A_2$;
- *or* $\exists s$ *s. t.* $\vdash A_1 : C \to s$ *and* $B =_\rho s$

Proof : by close inspection of the typing rules.

Our $(Appl)$ rule creates a redex in the type, so it is nearly always followed by a conversion rule. This implies that we will often have to check $\vdash B : s$ for the normalized type B, which makes the following correctness lemma crucial. It is called *weak* correctness because of the conclusion being only derivable in ρECC, whichever was the system of the assumption $\vdash A : B$.

The system ρECC finds its justification here, because we will need to exhibit a judgment $\vdash s : s'$ without knowing which sort is s, and this can be done only with an infinite hierarchy of universes. The $(ApplSort)$ rule is unavoidable too, because in the case of $(Appl)$, we could not use $(Appl)$ on the type $(B_1 \to s)\bullet A_2$ since we do not know how the redex $(B_1 \to s)\bullet A_2$ would be reduced.

Lemma 5.4 (Weak correctness)
For any terms A and B, $\vdash A : B \;\Rightarrow\; \exists s,\;\; \vdash_{\rho ECC} B : s$.

Proof : by induction on the derivation of $\vdash A : B$.

$(Axiom, Prod, ApplSort)$ With these rules $B \equiv s$, so $\exists s',\;\; \vdash_{\rho ECC} B \equiv s : s'$.

$(Start, Null, Abs, Conv)$ The judgment $\vdash B : s$ is in the assumptions.

$(Struct)$ $A \equiv A_1, A_2$ with $\vdash A_1 : B$ and $\vdash A_2 : B$. By induction hypothesis, we have $\vdash_{\rho ECC} B : s$.

$(Appl)$ $A \equiv A_1\bullet A_2$ and $B \equiv (B_1 \to B_2)\bullet A_2$, with $\vdash A_1 : B_1 \to B_2$, $\vdash B_1 : E$ and $\vdash A_2 : E$. The induction hypothesis on $\vdash A_1 : B_1 \to B_2$ gives $\vdash_{\rho ECC} B_1 \to B_2 : s$, thus by generation $\vdash_{\rho ECC} B_2 : s$ and $\vdash_{\rho ECC} B_1 : B_1' : s'$. The following derivation can then be established:

$$\dfrac{\dfrac{\dfrac{\vdash_{\rho ECC} B_1 : B_1' : s' \quad \vdash_{\rho ECC} s : s''}{\vdash_{\rho ECC} B_1 \to s : s''}\,(Prod) \qquad \vdash_{\rho ECC} B_2 : s}{\vdash_{\rho ECC} B_1 \to B_2 : B_1 \to s}\,(Abs) \qquad \vdash B_1 : E \quad \vdash A_2 : E}{\vdash_{\rho ECC} (B_1 \to B_2)\bullet A_2 : s}\,(ApplSort)$$

Remark 5.5 *To apply the $(Prod)$ rule here, we have to check that, if $s' \equiv \Box_i \;\wedge\; s'' \equiv \Box_j$, then $i \leq j$. But in any case we know that $\vdash s : s''$ so $s \equiv \Box_{j-1}$; on the other hand, we saw that $\vdash_{\rho ECC} B_1 \to B_2 : s$, thus $i \leq j - 1$, which trivially implies that $i \leq j$.*

□

For the subject reduction property, we need an additional constraint. When reducing a ρ-redex, if there are some distinct solutions to the corresponding matching problem, we will get a structure; looking at the $(Struct)$ rule, this means that the resulting term will be typable only if there is a common type for all the members of $\{\sigma C \mid \sigma \in \mathcal{S}ol(A \ll_{\mathbb{T}} B)\}$ (which is not obvious if a given σ affects the type of C). Let us note that this condition is trivially true if the theory is empty, since the matching is then unitary.

As for correctness, we have a *weak* property in the sense that the conclusion is derivable only in ρECC.

Theorem 5.6 (Weak subject reduction)
If the matching theory $\mathbb{T}$ *respects the following condition:*

$$\forall A, B, C, D,\ \vdash C : D \ \Rightarrow\ \exists D', s,\ \vdash D' : s\ \wedge\ \forall \sigma \in \mathcal{S}ol(A \ll_{\mathbb{T}} B),\ \sigma D =_\rho D'$$

Then typing judgments are preserved by reduction of the subject:

$$\forall A, B,\quad \vdash A : B\quad \wedge\quad A\ \mapsto_\rho\ A'\quad \Rightarrow\quad \vdash_{\rho ECC} A' : B$$

Proof : by induction on the derivation of $\vdash A : B$.

$(Axiom, Start, Null)$ A is in normal form so $\nexists A',\ A\ \mapsto_\rho\ A'$.

$(Struct, Abs, Prod)$ Here $A \equiv A_1, A_2$ ou $A \equiv A_1 \to A_2$, so $A\ \mapsto_\rho\ A'$ only if $A_1\ \mapsto_\rho\ A_1'$ or $A_2\ \mapsto_\rho\ A_2'$; the induction hypothesis then gives all the necessary assumptions to derive $\vdash_{\rho ECC} A' : B$.

$(Conv)$ The induction hypothesis can be applied on the assumption $\vdash A : B'$ to get $\vdash_{\rho ECC} A' : B'$, and then we just have to apply conversion as before to get $\vdash_{\rho ECC} A' : B$.

$(Appl)$ $A \equiv A_1 \bullet A_2$ and $B \equiv (B_1 \to B_2) \bullet A_2$, with $\vdash A_1 : B_1 \to B_2$, $\vdash B_1 : E$ and $\vdash A_2 : E$. We distinguish three cases:

- $\mathbf{A_1}\ \mapsto_\rho\ \mathbf{A_1'}$ or $\mathbf{A_2}\ \mapsto_\rho\ \mathbf{A_2'}$: as for $(Struct)$ or (Abs), we get the result by immediate application of the induction hypothesis.
- $\mathbf{A_1} \equiv \mathbf{null}$ or $\mathbf{A_1} \equiv \mathbf{C_1}, \mathbf{C_2}$: this a ν-redex (respectively δ-redex), so we just have to push the corresponding $(Null)$ (respectively $(Struct)$) rule at the beginning of the typing derivation.
- $\mathbf{A_1} \equiv \mathbf{C_1} \to \mathbf{C_2}$: this the real ρ-redex, with $A' \equiv \{\sigma C_2 \mid \sigma \in \mathcal{S}ol(C_1 \ll_{\mathbb{T}} A_2)\}$.
  By generation on $\vdash C_1 \to C_2 : B_1 \to B_2$, we know that $\exists s, D$ such that $\vdash C_2 : D$ and $B_1 \to B_2 =_\rho C_1 \to D$. If the equation $C_1 \ll_{\mathbb{T}} A_2$ has no solution, we can conclude immediately that $A' \equiv null$ and $\vdash_{\rho ECC} A' : B$. Elsewise, the condition on $\mathbb{T}$ ensures that there exists D' such that $\forall \sigma \in \mathcal{S}ol(C_1 \ll_{\mathbb{T}} A_2), \sigma D =_\rho D'$. Putting all these equalities together, we get:
  $(B_1 \to B_2) \bullet A_2 =_\rho (C_1 \to D) \bullet A_2 =_\rho \{\sigma D \mid \sigma \in \mathcal{S}ol(C_1 \ll_{\mathbb{T}} A_2)\} =_\rho D'$
  Then we can use the substitution lemma 5.2 to check that $\vdash \sigma C_2 : \sigma D$
  Using the correctness lemma for $(B_1 \to B_2) \bullet A_2$, we can deduce:

$$\dfrac{\dfrac{\vdash \sigma C_2 : \sigma D \quad \vdash_{\rho ECC} D' : s \quad \sigma D =_\rho D'}{\dfrac{\forall \sigma, \vdash_{\rho ECC} \sigma C_2 : D'}{\vdash_{\rho ECC} \{\sigma C_2\} : D'}\ (Struct)}\ (Conv) \qquad \vdash_{\rho ECC} (B_1 \to B_2) \bullet A_2 : s \quad (B_1 \to B_2) \bullet A_2 =_\rho D'}{\vdash_{\rho ECC} \{\sigma C_2\} : (B_1 \to B_2) \bullet A_2}\ (Conv)$$

Remark 5.7 *We are sure that the redex* $(C_1 \to D)\bullet A_2$ *is reducible whatever the strategy is, because* C_1 *and* A_2 *have appeared in the redex* $(C_1 \to C_2)\bullet A_2 (\equiv A)$*, which is known to be reducible.*

(*ApplSort*) This works very similarly to (*Appl*). When $\mathbf{A_1} \equiv \mathbf{C_1} \to \mathbf{C_2}$, we can deduce by generation that $\vdash_{\rho ECC} C_2 : s$, which makes things even easier: we just have to apply the substitution lemma.

$$\frac{\forall \sigma, \vdash_{\rho ECC} \sigma C_2 : \sigma s = s}{\vdash_{\rho ECC} \{\sigma C_2\} : s} \quad (Struct)$$

□

Corollary 5.8
This results is easily extended for the reflexive and transitive closure $\mapsto_\rho^*$ *, by induction on the number of occurences of* $\mapsto_\rho$ *in* $A \mapsto_\rho^* A'$*. If* $A \equiv A'$*, the result is trivial; elsewise we apply the induction hypothesis and the subject reduction theorem on the last reduction.*

Our last result for this section is the consistency of the calculus. The 'absurd' type is $\bot \stackrel{\triangle}{=} X^* \to X^*$, because if it is inhabited, any type is inhabited, just as in the λ-calculus. We require the terms not only to be closed, but also without constants because it would be easy to consider a constant $a^\bot$ for example. It is a kind of generalization of the notion of closure, because a constant is given with its type and therefore should appear in the "context". Finally, we forbid *null* too, since it admits any well-formed type, including of course $\bot$.

Lemma 5.9 (Consistency in the ρ-cube)
For any closed term A *in normal form, without constants and without null :*

$$\not\vdash A : \bot \quad (\bot \stackrel{\triangle}{=} X^* \to X^*)$$

Proof : by contradiction. Let us assume $\vdash A : \bot$ and distinguish cases on the structure of A:

$A \equiv s$: by generation, $\bot =_\rho s'$, which is impossible (both are in normal form and the calculus is confluent).

$A \equiv \alpha^B$: impossible since A is closed without constants.

$A \equiv A_1, A_2$: by generation, $\vdash A_1 : \bot$, so by inductively descending in the structure, we can find an A' which is not a structure and such that $\vdash A' : \bot$. The assumptions about A (normal form, closure, no constants) are obviously still true for A', and the length of A' is strictly less than the one of A, so this induction is obviously finite.

$A \equiv A_1 \bullet A_2$: in this case we are mostly interested in the head-application, so we write $A \equiv A_1 \bullet A_2 \bullet \ldots \bullet A_n$, where A_1 is no longer an application.

By closure property, we know that A_1 is not a variable nor a constant; A_1 can not be a sort s because the application $A_1 \bullet A_2$ is typable only if $\vdash A_1 : C \to D$ (but $\not\vdash s : C \to D$); A_1 can not be a structure, because $A_1 \bullet A_2$ would be a δ-redex, and thus A would not be in normal form.

We are left with the only possibility $A_1 \equiv B_1 \to B_2$, with $(B_1 \to B_2) \bullet A_2$ impossible to reduce because of the strategy. $A_1 \bullet A_2 \equiv (B_1 \to B_2) \bullet A_2$ is then typable (as a subterm of A), so we distinguish two different cases:

$n = 2$ so $A \equiv (B_1 \to B_2) \bullet A_2$, then $\vdash (B_1 \to B_2) \bullet A_2 : X^* \to X^*$. By generation, either $X^* \to X^* =_\rho s$ (impossible), or $X^* \to X^* =_\rho (B_1' \to B_2') \bullet A_2$ with $\vdash B_1 \to B_2 : B_1' \to B_2'$. Generation again gives then $B_1' \to B_2' =_\rho B_1 \to C_1$.

We have thus the following conversion relations : $X^* \to X^* =_\rho (B_1' \to B_2') \bullet A_2 =_\rho (B_1 \to C_1) \bullet A_2$. Applying the Church-Rosser property and remembering that $X^* \to X^*$ is in normal form, we can deduce that $(B_1 \to C_1) \bullet A_2 \mapsto_\rho^* X^* \to X^*$. This is only possible if the redex $(B_1 \to C_1) \bullet A_2$ is reducible; but then $(B_1 \to B_2) \bullet A_2$ is reducible too (as we assume that a strategy only considers the pattern and the argument). This is contradictory with the fact that A is in normal form.

$n \geq 3$: then the subterm $(B_1 \to B_2) \bullet A_2 \bullet A_3$ is typable, so $\exists C, D$ such that $\vdash (B_1 \to B_2) \bullet A_2 : C \to D$. But an analogous reasoning by generation and Church-Rosser gives $(B_1 \to C_1) \bullet A_2 \mapsto_\rho^* C \to D$; again we have proved that $(B_1 \to B_2) \bullet A_2$ is reducible, and so would be A.

Both cases have led to a contradiction: A can not be an application $A_1 \bullet A_2$.

$A \equiv A_1 \to A_2$: by generation, $\exists B_2, s$ such that $\vdash A_2 : B_2$, $\vdash A_1 \to B_2 : s$ and $A_1 \to B_2 =_\rho X^* \to X^*$.

By Church-Rosser, we deduce that $A_1 \to B_2 \mapsto_\rho^* X^* \to X^*$; but a reduction can not take place outside of A_1 or B_2, so we can decompose this into $A_1 \mapsto_\rho^* X^*$ and $B_2 \mapsto_\rho^* X^*$. The fact that A is in normal form implies that same for A_1 and B_2, so $A_1 \equiv X^*$ and $B_2 \equiv X^*$. Let us then distinguish cases on A_2, knowing that $\vdash A_2 : X^* \ (\equiv B_2)$:

$A_2 \equiv s$: then $\vdash s : X^*$, impossible.

$A_2 \equiv \alpha$: by closure of A, as $A_1 \equiv X^*$, α can only be X^*. This gives the typing judgment $\vdash X^* : X^*$, which is impossible considering the typing rule $(Start)$ and the fact that $X^* \neq_\rho *$.

$A_2 \equiv C_1, C_2$: the same reasoning remains valid with C_1 instead of A_2; since it is shorter, we can not do it indefinitely.

$A_2 \equiv C_1 \to C_2$: we know that $\vdash A_2 : X^*$. Generation then gives $X^* =_\rho s$ or $X^* =_\rho C_1 \to C_2'$, both being obviously impossible. A_2 can not be an abstraction.

$A_2 \equiv C_1 \bullet C_2$: a similar reasoning to that conducted for $A \equiv A_1 \bullet A_2$ can be led: the only possible case is that C_1 is the variable X^*, which would not break the closure property since $BV(A) = \{X^*\}$. We have thus $A_2 \equiv X^* \bullet C_2 \bullet \ldots \bullet C_n$.

Then A_2 is typable only if $X^* \bullet C_2$ is typable, but it is impossible to assign an "arrow" type to X^*. It follows that A_2 can not be an application.
All cases have been proved to be impossible !

□

Remark 5.10 *The previous lemma remains valid for a weakly normalizing term A: we begin with finding a normal form A′, then apply subject reduction; consistency applied to A′ immediately implies consistency for A. However, it is crucial that the additional assumption (on the theory) for subject reduction is verified.*

Let us note that strong normalization and decidability have not been dealt with. A promising research direction for normalization is to "compile" the ρ-calculus in a former language known to be normalizing, as the calculus of constructions for example.

It seems reasonable to guess that decidability then follows in a similar way as for the λ-calculus. However, one has to be cautious with the matching theory $\mathbb{T}$: it is quite obvious that the typing judgments (and the calculus itself !) become undecidable if the equality $\overset{\mathbb{T}}{=}$ is not decidable.

6 Results and issues about the uniqueness of typing

First we show uniqueness failure on counterexamples, and we prove that the number of distinct types for a given term is unbounded but finite. Positive results about uniqueness are given for $\rho 2$ and $\rho\!\rightarrow$.

6.1 Uniqueness failures

As shown in [11], in the system ρP, the term $X^{Y^*} \rightarrow Y^*$ can be considered either as an abstraction (the "type extractor") or as a type:

$$\dfrac{\dfrac{\vdash X^{Y^*} : Y^* : * \quad \vdash * : \Box}{\vdash Y^* : * \qquad \vdash X^{Y^*} \rightarrow * : \Box}\ (Prod\ *, \Box)}{\vdash X^{Y^*} \rightarrow Y^* : X^{Y^*} \rightarrow *}\ (Abs) \qquad\qquad \dfrac{\vdash X^{Y^*} : Y^* : * \quad \vdash Y^* : *}{\vdash X^{Y^*} \rightarrow Y^* : *}\ (Prod\ *, *)$$

Of course these derivations are correct too in $\rho P\underline{\omega}$, $\rho P2$, ρCC and ρECC as they all extend ρP.

Uniqueness is no longer valid $\rho\omega = \rho 2 + \rho\underline{\omega}$, because depending on the subsystem in which we derive the judgment, $X^* \rightarrow X^*$ is an abstraction (the "identity on types") or a type (the absurd type $\perp$):

$$\dfrac{\dfrac{\vdash X^* : * : \Box \quad \vdash * : \Box}{\vdash X^* : * \qquad \vdash X^* \to * : \Box}\ (Prod\ \Box, \Box)}{\vdash X^* \to X^* : X^* \to *}\ (Abs) \qquad\qquad \dfrac{\vdash X^* : * : \Box \quad \vdash X^* : *}{\vdash X^* \to X^* : *}\ (Prod\ \Box, *)$$

Let us notice that two types are distinct if they are not convertible to each other. In these two cases, it is easy to see that the considered types are in normal form, so by application of the Church-Rosser property, they are convertible only if they are syntactically equivalent, which is obviously not the case.

These two examples can be extended to find terms with an arbitrary number of types n:

$$\vdash_{\rho P} X_1^{Y_1^*} \to X_2^{Y_2^*} \to \ldots \to X_n^{Y_n^*} \to Y_n^* : \begin{cases} * \\ X_1^{Y_1^*} \to * \\ \ldots \\ X_1^{Y_1^*} \to X_2^{Y_2^*} \to \ldots \to X_n^{Y_n^*} \to * \end{cases}$$

$$\vdash_{\rho\omega} X_1^* \to X_2^* \to \ldots \to X_n^* \to X_n^* : \begin{cases} * \\ X_1^* \to * \\ \ldots \\ X_1^* \to X_2^* \to \ldots \to X_n^* \to * \end{cases}$$

Fig. 6. Terms admitting $n+1$ distinct types (in ρP and in $\rho\omega$

However, we can show that a given term always has a finite number of distinct types. We must still exclude *null* since it can give any well-formed type, and there is an infinity of them.

Lemma 6.1 (Typing finiteness)
Any ρ-term without null admits only a finite number of distinct types (modulo $=_\rho$).

Proof : by induction on the structure of A. Since conversion only replaces a type with another which is convertible to it, we will "skip" the conversions and consider that all the typing judgments are ended by the rule that corresponds to the syntax of the term A.

$A \equiv s$ By generation, all the types of A are convertible to the only s' such that $\vdash s : s'$.

$A \equiv \alpha^B$ By generation, all the types of A are convertible to B.

$A \equiv A_1, A_2$ By induction hypothesis, both A_1 and A_2 have a finite number of distinct types. As A can only admit types that are correct for A_1 *and* A_2, the typing is finite for A too.

$A \equiv A_1 \to A_2$ The two rules (Abs) and $(Prod)$ could have been used, so we show finiteness for each case; then by union of these sets of types, we will get a finite set which will contain all the admissible types for A:

- If $\vdash A_1 \to A_2 : s$, then by generation $\vdash A_2 : s$; by induction hypothesis, there is only a finite number of possible s which fit.
- If $\vdash A_1 \to A_2 : A_1 \to B_2$, we only have to prove that there is a finite number of possible B_2; but by generation, $\vdash A_2 : B_2$ so by induction hypothesis, we get the wanted result.

$A \equiv A_1 \bullet A_2$ Here again, we prove finiteness for each case:

- If $\vdash A_1 \bullet A_2 : (C \to D) \bullet A_2$, then $\vdash A_1 : C \to D$. By induction hypothesis, there is only a finite set of possible $C \to D$, and since $A \equiv A_1 \bullet A_2$ is fixed, this the only part that can vary in $(C \to D) \bullet A_2$.
- If $\vdash A_1 \bullet A_2 : s$, then $\vdash A_1 : C \to s$. By induction hypothesis, there is a finite number of admissible $C \to s$, so of course the set of possible s is finite too.

□

6.2 Uniqueness in ρ2

This is the main result of this paper. Subject reduction is needed, so we make the appropriate assumptions on the matching theory; as usual, *null* is excluded because it would introduce an uncontrolled set of distinct types.

Theorem 6.2 (Typing uniqueness in $\rho 2$)
With a theory $\mathbb{T}$ compatible with subject reduction, every ρ-term A without null is such that:

$$\vdash_{\rho 2} A : B \quad \wedge \quad \vdash_{\rho 2} A : B' \quad \Rightarrow \quad B =_\rho B'$$

Proof : by induction on the structure of A. Whenever two distinct types are possible, we show that we can not be in $\rho 2$.

$A \equiv s$: by generation, $B =_\rho s' =_\rho B'$, where s' is the only sort such that $\vdash_{\rho 2} s : s'$.

$A \equiv \alpha^C$: generation directly gives $B =_\rho C =_\rho B'$.

$A \equiv A_1, A_2$: by generation, $\vdash_{\rho 2} A_1 : B$ and $\vdash_{\rho 2} A_1 : B'$, so by induction hypothesis $B =_\rho B'$.

$A \equiv A_1 \bullet A_2$: let us show that the last applied rule can not be $(ApplSort)$. If there exist C, s such that $\vdash_{\rho 2} A_1 : C \to s$, a careful inspection of the rules (Abs), $(Start)$ and $(Conv)$ shows that $\vdash_{\rho 2} C \to s : s'$ (still in $\rho 2$!) By generation on this last typing judgment, we get $\vdash_{\rho 2} s : s'$, which implies (in $\rho 2$) that $s \equiv *$ and $s' \equiv \Box$:

$$\dfrac{\dfrac{\dfrac{\vdash_{\rho 2} C : C' : s_1 \quad \vdash_{\rho 2} * : \Box}{\vdash_{\rho 2} C \to * : \Box}\ (s_1, \Box) \qquad \ldots}{\vdash_{\rho 2} A_1 : C \to *}\ (\ldots) \qquad \vdash_{\rho 2} C : E \quad \vdash_{\rho 2} A_2 : E}{\vdash_{\rho 2} A_1 \bullet A_2 : *}\ (ApplSort)$$

But then there is a $(Prod)$ rule with the pair $(s_1, \Box)$, which is impossible in $\rho 2$: the only available product rules are $(\Box, *)$ et $(*, *)$.

Now we know that the last applied rule was $(Appl)$, so by generation, $\exists C, D, C', D'$ such that $\vdash_{\rho 2} A_1 : C \to D$ and $\vdash_{\rho 2} A_1 : C' \to D'$. By induction hypothesis on A_1, we get $C \to D =_\rho C' \to D'$, so we can conclude with the following conversions: $B =_\rho (C \to D) \bullet A_2 =_\rho (C' \to D') \bullet A_2 =_\rho B'$.

$A \equiv A_1 \to A_2$: the last applied rule can be (Abs) or $(Prod)$, so we must deal with the cases when they are the same for B as for B', and when they are different.

- If $\exists C, C'$ such that $\vdash_{\rho 2} A_2 : C$ and $\vdash_{\rho 2} A_2 : C'$, then by induction hypothesis $C =_\rho C'$, so $B =_\rho A_1 \to C =_\rho A_1 \to C' =_\rho B'$.
- If $\exists s_2, s_2'$ such that $\vdash_{\rho 2} A_2 : s_2$ and $\vdash_{\rho 2} A_2 : s_2'$, then by induction hypothesis $s_2 =_\rho s_2'$, so $B =_\rho s_2 =_\rho s_2' =_\rho B'$.
- When different rules have been applied, we must detail a little more the inferences, like in the following figure. On the one hand, we have $\exists s, C$ such that $\vdash_{\rho 2} A_2 : C$, $\vdash_{\rho 2} A_1 \to C : s$ and $B =_\rho A_1 \to C$; on the other hand $\exists s_1, s_2, D$ with $\vdash_{\rho 2} A_1 : D : s_1$, $\vdash_{\rho 2} A_2 : s_2$ et $B =_\rho s_2$.

$$\dfrac{\vdash_{\rho 2} A_2 : C \qquad \dfrac{\vdash_{\rho 2} A_1 : A_1' : s' \quad \vdash_{\rho 2} C : s}{\vdash_{\rho 2} A_1 \to C : s}\ (s', s)}{\vdash_{\rho 2} A_1 \to A_2 : A_1 \to C}\ (Abs) \qquad \dfrac{\vdash_{\rho 2} A_1 : D : s_1 \quad \vdash_{\rho 2} A_2 : s_2}{\vdash_{\rho 2} A_1 \to A_2 : s_2}\ (s_1, s_2)$$

By induction hypothesis on A_2, we know that $C =_\rho s_2$. But generation applied to $\vdash_{\rho 2} A_1 \to C : s$ ensures that $\vdash_{\rho 2} C : s$, thus by subject reduction we know that $\vdash_{\rho ECC} s_2 : s$.

It follows that $s_2 \equiv *$ and $s \equiv \Box$ (both s and s_2 have appeared in derivations in $\rho 2$, so they can not be $\Box_i$). But then the type inference with (Abs) (on the left) uses a product rule (s', s) : since we have proved that $s \equiv \Box$, this is impossible in $\rho 2$ (the only available product rules are $(\Box, *)$ et $(*, *)$).

□

Corollary 6.3 *Typing uniqueness is valid in $\rho\to$ too, because it is a subsystem of $\rho 2$: any typing judgment in $\rho\to$ can be straightforwardly translated in $\rho 2$, so if a term had distinct types in $\rho\to$ it would admit them in $\rho 2$ too.*

This result is particularly interesting when looking at $\rho 2$ as an extended model of ML. In fact, since it includes *explicit* polymorphism and very general patterns, it is much more powerful than ML, which has only external quantification on type variables, and linear patterns. For instance, the term $Y^* \to X^{Y^*} \to X^{Y^*}$ corresponds to the program `fun id x = x`, which type is `'a -> 'a`. We can notice that ML does not take the type `'a` as an argument, but infers it at a meta-level by looking at the type of the argument which will be passed to `id`.

Typing is fundamental in ML, since a function is accepted only if it is well typed. But the typechecking mechanism (algorithm W of Damas-Milner) is quite basic: it just checks that the types of a function and its arguments are coherent. In particular, it totally discards all the pattern matching aspects; here we have defined types with reductions, which could be expected to be more "precise" about the behavior of a function. Our results are thus more general, and could be specialized for MLby making appropriate restrictions.

Uniqueness of typing is also an interesting result for a programming language with strong typing, as ML. It ensures that, whatever the algorithm and the implementation for type inference are, the behavior of the language will be the same, hence satisfying a certain notion of (desirable !) determinism.

7 Conclusions

We have studied the typed aspects of the ρ-calculus, which integrates both the λ-calculus and the rewriting in one formalism. The defined type systems were mainly adapted from the systems of Barendregt's λ-cube, and from the Extended Calculus of Constructions of Luo. The main innovations are: abstraction on an arbitrary pattern, choice of a matching theory, treatment of the resulting non-determinism, unification of $\lambda$and Π in a single abstractor.

We have seen that some adjustments are necessary: strategies for recovering of confluence, typing of the substitutions. Under these assumptions, we have proved most of the classical properties about typed calculi: stability up to substitution, correctness of types, subject reduction, consistency. Some of these properties require the introduction of the system ρECC with an infinite hierarchy of universes. Uniqueness of typing is no longer valid: because of some ambiguity between abstractions and products, the number of distinct types for a given term is unbounded but finite. However, uniqueness remains true in $\rho 2$, the polymorphic ρ-calculus, which embeds ML quite naturally.

8 Future work

We are now interested in finding further results about typing:

- bounds on the number of distinct types of a term;
- ordering of these types, and existence of a principal type;

- conditions for uniqueness in other systems than $\rho 2$ (for example, what happens if we add the constraint $\vdash A : B : *$);
- weak and strong normalization of the typed calculus (which would imply consistency and probably decidability).

José Meseguer [24,25] has shown how to encode logics with rewriting mechanisms; we would like to study the ρ-calculus as a proof-term language, in a Curry-Howard fashion. As shown in [7], patterns are a natural way of modeling logical formulas, and we would like to understand what are the logical applications of matching modulo a theory, non determinism and structures. The non-uniqueness of typing can also be very interesting, because if it is true at the level of terms, it will mean that a given proof is correct for two distinct propositions, which has to be understood and exploited.

References

[1] H. Ait-Kaci, A. Podelski, and G. Smolka. A feature constraint system for logic programming with entailment. *Theoretical Computer Science*, 122(1-2):263–283, 1994.

[2] Henk Barendregt. *Handbook of Logic in Computer Science*, chapter Lambda Calculi with Types, pages 117–309. Clarendon Press, 1992.

[3] Gilles Barthe, Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. Pure patterns type systems. To be published, 2002.

[4] Clara Bertolissi. Traduction des Combinatory Reduction Systems en ρ-calcul. Mémoire de dea, LORIA - INPL, Nancy, 2002.

[5] F. Blanqui. *Théorie des Types et Réécriture (Type Theory and Rewriting)*. PhD thesis, Université Paris-Sud (France), 2001. Available at `http://www.lri.fr/~blanqui`. An english version will be available soon.

[6] Roel Bloo, Fairouz Kamareddine, and Rob Nederpelt. On Pi-conversion in the lambda-cube and the combination with abbreviations. *Annals of Pure and Applied Logic*, 97(1 – 3):27 – 45, 1999.

[7] Serenella Cerrito and Delia Kesner. Pattern matching as cut elimination. In Giuseppe Longo, editor, *Fourteenth Annual IEEE Symposium on Logic in Computer Science*, Trento, Italy, July 1999. IEEE Comp. Soc. Press.

[8] Horatiu Cirstea. *Calcul de réécriture : fondements et applications.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, 2000.

[9] Horatiu Cirstea. Stratégies confluentes pour le ρ-calcul. Communication personnelle, 2001.

[10] Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. Matching Power. In Aart Middeldorp, editor, *Rewriting Techniques and Applications*, volume 2051 of *Lecture Notes in Computer Science*, Utrecht, The Netherlands, May 2001. Springer-Verlag.

[11] Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. The Rho Cube. In Furio Honsell, editor, *Foundations of Software Science and Computation Structures*, volume 2030 of *Lecture Notes in Computer Science*, pages 166–180, Genova, Italy, April 2001.

[12] Thierry Coquand. Pattern matching with dependent types. In Bengt Nordström, Kent Petersson, and Gordon Plotkin, editors, *Informal proceedings workshop on types for proofs and programs*, pages 71 – 84. Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg Univ., Båstad, Suède, June 1992.

[13] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95 – 120, 1988.

[14] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, June 1972.

[15] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[16] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Second Symposium of Logic in Computer Science*, pages 194 – 204, Ithaca, N. Y., 1987. IEEE, Washington DC.

[17] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, 1982.

[18] G. Huet. *Résolution d'equations dans les langages d'ordre 1,2, ...,ω*. Thèse de Doctorat d'Etat, Université de Paris 7 (France), 1976.

[19] Fairouz Kamareddine and Rob Nederpelt. Canonical typing and π-conversion in the Barendregt cube. *Journal of Functional Programming*, 6(2):245 – 267, 1996.

[20] Delia Kesner, Laurence Puel, and Val Tannen. A typed pattern calculus. *Information and Computation*, 124(1):32–61, 10 January 1996.

[21] Jan Willem Klop. Term rewriting systems. In Samson Abramsky, Dov Gabbay, and Tom Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Clarendon, 1992.

[22] Donald E. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.

[23] Z. Luo. ECC: An Extended Calculus of Constructions. In *Proceedings of LICS*, pages 385–395, 1990.

[24] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Menlo Park, California, August 1993.

[25] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[26] G. Plotkin. Call by name, call by value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[27] Morten Heine Sørensen and Pawel Urzyczyn. Lectures on the curry-howard isomorphism. Lecture Notes 98/14, DIKU, Copenhagen, 1998.

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