Electronic Notes in Theoretical Computer Science 71 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume71.html 17 pages

Rewriting Calculus with(out) Types

Horatiu Cirstea, Claude Kirchner, and Luigi Liquori

LORIA, INRIA and University of Nancy 2
Campus Scientifique, BP 239, 54506 Vandoeuvre-lès-Nancy Cedex France
{Horatiu.Cirstea,Claude.Kirchner,Luigi.Liquori}@loria.fr

Abstract

The last few years have seen the development of a new calculus which can be considered as an outcome of the last decade of various researches on (higher order) term rewriting systems, and lambda calculi. In the Rewriting Calculus (or Rho Calculus, ρCal), algebraic rules are considered as sophisticated forms of "lambda terms with patterns", and rule applications as lambda applications with pattern matching facilities.

The calculus can be customized to work modulo sophisticated theories, like commutativity, associativity, associativity-commutativity, etc. This allows us to encode complex structures such as list, sets, and more generally objects. The calculus can either be presented "à la Curry" or "à la Church" without sacrificing readability and without complicating too much the metatheory.

Many static type systems can be easily plugged-in on top of the calculus in the spirit of the rich type-oriented literature. The Rewriting Calculus could represent a *lingua franca* to encode many paradigms of computations together with a formal basis used to build powerful theorem provers based on lambda calculus and efficient rewriting, and a step towards new proof engines based on the Curry-Howard isomorphism.

1 Introduction

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.

Pattern matching occurs implicitly in many languages through the simple parameter passing mechanism, and explicitly in languages like ML and Prolog, where it can be quite sophisticated [29]. It is somewhat astonishing that one of the most commonly used model of computation, the lambda calculus, uses only trivial pattern matching. This has been extended, initially for programming concerns, either by the introduction of patterns in lambda calculi [34,36], or by the introduction of matching and rewrite rules in functional programming languages. And indeed, many works address the integration of term rewriting with lambda calculus, either by enriching first-order rewriting with higher-order capabilities [26,37,31], or by adding to lambda calculus algebraic features allowing one, in particular, to deal with equality in an efficient way [32,4,19,23].

Pattern abstractions generalize lambda abstractions by binding structured expressions instead of variables, and are commonly used to compile case-expressions in functional programming languages [34] and to provide term calculi for sequent calculi [25]. For example, the pattern abstractions $\lambda 0.0$ and $\lambda \mathsf{succ}(\mathcal{X}).\mathcal{X}$ are used to compile the predecessor function $\lambda \mathcal{X}.\ \mathsf{case}\ \mathcal{X}\ \mathsf{of}\{0 \twoheadrightarrow 0 \mid \mathsf{succ}(\mathcal{X}) \twoheadrightarrow \mathcal{X}\}$, whereas the pattern abstraction $\lambda\langle\mathcal{X},\mathcal{Y}\rangle.\ \mathcal{X}$ is used to encode the sequent derivation

$$\dfrac{\dfrac{\sigma,\tau \vdash \sigma}{\sigma \wedge \tau \vdash \sigma}}{\vdash (\sigma \wedge \tau) \to \sigma}$$

Rule abstractions generalize in turn pattern abstractions by binding arbitrary expressions instead of patterns, and are used in the Rewriting Calculus to provide a first-class account of rewrite rules and rewriting strategies. For example, rule abstractions can be used to encode innermost rewriting strategies for term rewriting systems. Furthermore, rule abstractions correspond to a form of higher-order natural deduction, where (parts of) proof trees are discharged instead of assumptions. Although such rule abstractions are a firmly grounded artifact both in logic and in programming language design and implementation, they lack established foundations.

In the Rewriting Calculus, the usual lambda abstraction $\lambda\mathcal{X}.\mathcal{T}$ is replaced by a rule abstraction $\mathcal{T}_1 \twoheadrightarrow \mathcal{T}_2$, where $\mathcal{T}_1$ is an arbitrary term (in jargon a *pattern*) and $\mathcal{T}_2$ is the argument to be fired, where the free variables of $\mathcal{T}_1$ are bound (via pattern matching) in $\mathcal{T}_2$.

The application of an abstraction $\mathcal{T}_1 \twoheadrightarrow \mathcal{T}_2$ to a term $\mathcal{T}_3$ always "fires" and produces as result the term $[\mathcal{T}_1 \ll \mathcal{T}_3]\mathcal{T}_2$ which represents a *delayed matching constraint*, *i.e.* a term where the matching equation is "put on the stack". The matching constraint will be (self) evaluated (if a matching solution exists) or delayed (if no solution exists). If a solution σ exists, the delayed matching constraint self-evaluates to $\sigma(\mathcal{T}_3)$.

The presentations is intentionally kept informal, with few definitions, no appendix, and no proofs; to stimulate the curious reader, some exercises are given. We adopt the simplest algebraic theory we know off: the syntactic one. Quite simply, the main aim of this paper is to introduce, explain, provide examples, and stimulate the reader to hear more about the Rewriting Calculus [8,9,7,10].

2 Syntax

The untyped (*i.e.* à la Curry) syntax of the Rho Calculus (ρCal) extends the one presented in [11,12] by adding delayed matching constraints. As in any calculus involving binders, we work modulo the "*α-convention*" of Church [6], and modulo the "*hygiene-convention*" of Barendregt [1], *i.e.* free and bound variables have different names. The symbol $\equiv$ denotes syntactic identity of objects like terms or substitutions. The syntax of ρCal is defined as follows:

$$\mathcal{T} ::= \mathcal{X} \mid \mathcal{K} \mid \mathcal{T} \twoheadrightarrow \mathcal{T} \mid [\mathcal{T} \ll \mathcal{T}]\mathcal{T} \mid \mathcal{T}\bullet\mathcal{T} \mid \mathcal{T}, \mathcal{T}$$

where $\mathcal{X}$ represents the set of variables and $\mathcal{K}$ the set of constants. By abuse of notation, we denote members of these sets by the same letters possibly indexed. Notice that there are two kinds of "abstractions":

(i) $\mathcal{T}_1 \twoheadrightarrow \mathcal{T}_2$ denotes a *rule abstraction* with pattern $\mathcal{T}_1$ and body $\mathcal{T}_2$; the free variables of $\mathcal{T}_1$ are bound in $\mathcal{T}_2$.

(ii) $[\mathcal{T}_1 \ll \mathcal{T}_2]\mathcal{T}_3$ denotes a *delayed matching constraint* with pattern $\mathcal{T}_1 \ll \mathcal{T}_2$ and body $\mathcal{T}_3$; the free variables of $\mathcal{T}_1$ are bound in $\mathcal{T}_3$ but not in $\mathcal{T}_2$.

To support the intuition, we mention here that the application of an abstraction $\mathcal{T}_1 \twoheadrightarrow \mathcal{T}_3$ to a term $\mathcal{T}_2$ always "fires" and produces as result the term $[\mathcal{T}_1 \ll \mathcal{T}_2]\mathcal{T}_3$ which represents a constrained term where the matching equation is "put on the stack". The body of the constrained term will be evaluated or delayed according to the result of the corresponding matching problem. If a solution exists, the delayed matching constraint self-evaluates to $\sigma(\mathcal{T}_3)$, where σ is the solution of the matching between $\mathcal{T}_1$ and $\mathcal{T}_2$. Finally, as a sort of syntactic sugar, and directly inspired from previous works on the ρCal, terms can be grouped together into *structures* built using the symbol ",".

We assume that the (very often hidden) application operator "$\bullet$" associates to the left, while the other operators associate to the right. The priority of "$\bullet$" is higher than that of "$[\_ \ll \_]\_$" which is higher than that of the "$\twoheadrightarrow$" which is, in turn, of higher priority than the ",".

Definition 2.1 [Signatures and Abbreviations]

$\mathcal{T}_1.\mathcal{T}_2$	$\triangleq$	$\mathcal{T}_1 \ \mathcal{T}_2 \ \mathcal{T}_1$	self-application
$\mathcal{T}_0(\mathcal{T}_1 \cdots \mathcal{T}_n)$	$\triangleq$	$\mathcal{T}_0 \ \mathcal{T}_1 \ \cdots \ \mathcal{T}_n$	function-application
$(\mathcal{T}_i)^{i=1\ldots n}$	$\triangleq$	$\mathcal{T}_1, \cdots, \mathcal{T}_n$	structure

We draw the attention of the reader on the main difference between "$\bullet$" denoting the *application* operator, and "." denoting the object-oriented *self-application* operator of S. Kamin [24].

3 Matching

Before introducing the main machinery underneath ρCal, *i.e.* matching, we adapt the classical notion of simultaneous substitution application to deal with the new forms of constrained terms introduced in the ρCal.

Definition 3.1 [Substitutions]
A substitution σ is a mapping from the set of variables to the set of terms. A finite substitution has the form $\{\mathcal{T}_1/\mathcal{X}_1 \ldots \mathcal{T}_m/\mathcal{X}_m\}$; the empty substitution $\{\,\}$ is denoted by σ_{ID}. The application of a substitution σ to a term $\mathcal{T}$, denoted as usual by $\sigma(\mathcal{T})$, is defined as follows:

$$
\begin{array}{llllll}
\sigma_{\mathsf{ID}}(\mathcal{T}) & \triangleq & \mathcal{T} & \sigma(\mathcal{T}_1 \twoheadrightarrow \mathcal{T}_2) & \triangleq & \mathcal{T}_1 \twoheadrightarrow \sigma(\mathcal{T}_2) \\
\sigma(\mathcal{K}) & \triangleq & \mathcal{K} & \sigma([\mathcal{T}_1 \ll \mathcal{T}_2]\mathcal{T}_3) & \triangleq & [\mathcal{T}_1 \ll \sigma(\mathcal{T}_2)]\sigma(\mathcal{T}_3) \\
\sigma(\mathcal{X}_i) & \triangleq & \begin{cases} \mathcal{T}_i & \text{if } \mathcal{X}_i \in Dom(\sigma) \\ \mathcal{X}_i & \text{otherwise} \end{cases} & \begin{array}{l} \sigma(\mathcal{T}_1\, \mathcal{T}_2) \\ \sigma(\mathcal{T}_1, \mathcal{T}_2) \end{array} & \begin{array}{l} \triangleq \\ \triangleq \end{array} & \begin{array}{l} \sigma(\mathcal{T}_1)\ \sigma(\mathcal{T}_2) \\ \sigma(\mathcal{T}_1), \sigma(\mathcal{T}_2) \end{array}
\end{array}
$$

This defines higher-order substitutions as we work modulo α-*convention*; when applying a substitution to an abstraction, we know that the free variables of the corresponding abstracted pattern do not belong to the domain of the substitution. This definition is compatible with the evaluation mechanism of the ρCal.

In ρCal, we deal with abstractions on patterns and their application; thus, computing the substitution which solves the matching from a pattern $\mathcal{T}_1$ to a subject $\mathcal{T}_2$ is a crucial ingredient of the calculus. We focus here on *syntactic matching* and we revisit the corresponding notions and algorithms. We denote by Δ a list of variables and we use the following definition for syntactic matching:

Definition 3.2 [Matching Equations and Solutions]

(i) A *match equation* is a formula of the form $\mathcal{T}_1 \prec\!\!\prec_\Delta \mathcal{T}_2$;

(ii) A *matching system* $\mathsf{T} \triangleq \bigwedge_{i=0\ldots n} \mathcal{T}_i \prec\!\!\prec_{\Delta_i} \mathcal{T}_i'$ is a conjunction of match equations, where $\wedge$ is associative and commutative;

(iii) A matching system T is "successful" if it is empty or:
  (a) has the shape: $\bigwedge_{i=0\ldots n} \mathcal{X}_i \prec\!\!\prec_{\Delta_i} \mathcal{T}_i \bigwedge_{j=0\ldots m} \mathcal{K}_j \prec\!\!\prec_{\Delta_j} \mathcal{K}_j$;
  (b) for all $h, k = 0 \ldots n$: $\mathcal{X}_h \equiv \mathcal{X}_k$ implies $\mathcal{T}_h \equiv \mathcal{T}_k$;
  (c) for all $i = 0 \ldots n$: $\mathcal{X}_i \in \Delta_i$ implies $\mathcal{X}_i \equiv \mathcal{T}_i$;
  (d) for all $i = 0 \ldots n$, $FV(\mathcal{T}_i) \cap \Delta_i \neq \emptyset$ implies $\mathcal{X}_i \equiv \mathcal{T}_i$.

(iv) The substitution $\sigma_{\mathsf{T}} \triangleq \{\mathcal{T}_1/\mathcal{X}_1 \cdots \mathcal{T}_n/\mathcal{X}_n\}$ is the solution of a successful matching system T.

The engine underneath computations in ρCal solves matching constraints,

$$(\rho)\ \ (\mathcal{T}_1 \twoheadrightarrow \mathcal{T}_2)\mathcal{T}_3 \quad \to_\rho \quad [\mathcal{T}_1 \ll \mathcal{T}_3]\mathcal{T}_2$$

$$(\sigma)\ \ [\mathcal{T}_1 \ll \mathcal{T}_3]\mathcal{T}_2 \quad \to_\sigma \quad \sigma_{(\mathcal{T}_1 \ll \mathcal{T}_3)}(\mathcal{T}_2)$$

$$(\delta)\ \ (\mathcal{T}_1, \mathcal{T}_2)\, \mathcal{T}_3 \quad \to_\delta \quad \mathcal{T}_1\, \mathcal{T}_3, \mathcal{T}_2\, \mathcal{T}_3$$

Fig. 1. Small-step reduction semantics

i.e. given a delayed matching constraint $[\mathcal{T}_1 \ll \mathcal{T}_2]\mathcal{T}_3$, find a solution σ of the matching equation $\mathcal{T}_1 \prec\!\!\prec_\emptyset \mathcal{T}_2$ (if it exists), and continue the computation with $\sigma(\mathcal{T}_3)$.

Definition 3.3 [Matching Algorithm $\prec\!\!\prec_\Delta$]
The matching substitution solving a matching equation can be computed by the following *matching reduction system*, where $\Delta' = FV(\mathcal{T}_1)$:

$$
\begin{array}{lll}
(\mathit{Rule}) & (\mathcal{T}_1 \twoheadrightarrow \mathcal{T}_2) \prec\!\!\prec_\Delta (\mathcal{T}_1 \twoheadrightarrow \mathcal{T}_3) & \leadsto\ \mathcal{T}_2 \prec\!\!\prec_{\Delta,\Delta'} \mathcal{T}_3 \\
(\mathit{Delay}) & [\mathcal{T}_1 \ll \mathcal{T}_2]\mathcal{T}_4 \prec\!\!\prec_\Delta [\mathcal{T}_1 \ll \mathcal{T}_3]\mathcal{T}_5 & \leadsto\ \mathcal{T}_2 \prec\!\!\prec_\Delta \mathcal{T}_3 \wedge \mathcal{T}_4 \prec\!\!\prec_{\Delta,\Delta'} \mathcal{T}_5 \\
(\mathit{Appl}) & (\mathcal{T}_1\, \mathcal{T}_2) \prec\!\!\prec_\Delta (\mathcal{T}_3\, \mathcal{T}_4) & \leadsto\ \mathcal{T}_1 \prec\!\!\prec_\Delta \mathcal{T}_3 \wedge \mathcal{T}_2 \prec\!\!\prec_\Delta \mathcal{T}_4 \\
(\mathit{Struct}) & (\mathcal{T}_1, \mathcal{T}_2) \prec\!\!\prec_\Delta (\mathcal{T}_3, \mathcal{T}_4) & \leadsto\ \mathcal{T}_1 \prec\!\!\prec_\Delta \mathcal{T}_3 \wedge \mathcal{T}_2 \prec\!\!\prec_\Delta \mathcal{T}_4
\end{array}
$$

Starting from a matching equation $\mathcal{T}_1 \prec\!\!\prec_\emptyset \mathcal{T}_2$, the application of this rule set obviously terminates and either leads to an unsuccessful matching system in which case we say that the matching has failed or a substitution σ such that $\sigma(\mathcal{T}_1) \equiv \mathcal{T}_2$ is exhibited. We denote such a substitution by $\sigma_{(\mathcal{T}_1 \ll \mathcal{T}_2)}$.

Exercise. If you love matching algorithms:

(i) complete the algorithm $\prec\!\!\prec_\Delta$ by adding the failure symbol F and enumerating all the possible reduction rules dealing with matching failures, that would lead to reductions like $f(3) \prec\!\!\prec_\Delta f(4) \leadsto \mathsf{F}$;

(ii) customize the algorithm $\prec\!\!\prec_\Delta$ with more sophisticated algebraic theories, *e.g.* the one where the "," operator is associative and (or) commutative (hint: check the J.-M. Hullot's or S. Eker's algorithms [22,15]).

4 Semantics

4.1 Small-step Reduction Semantics

The small-step reduction semantics is defined by the reduction rules presented in Figure 1. The central idea of the (ρ) rule of the calculus is that the application of a term $\mathcal{T}_1 \twoheadrightarrow \mathcal{T}_2$ to a term $\mathcal{T}_3$, reduces to the delayed matching constraint $[\mathcal{T}_1 \ll \mathcal{T}_3]\mathcal{T}_2$, while (the application of) the (σ) rule consists in solving the matching equation $\mathcal{T}_1 \prec\!\!\prec_{\emptyset} \mathcal{T}_3$, and applying the obtained result to the the term $\mathcal{T}_2$. The rule (δ) deals with the distributivity of the application on the structures built with the "," constructor.

It is important to remark that if $\mathcal{T}_1$ is a variable, then the subsequent combination of (ρ) and (σ) rules corresponds exactly to the (β) rule of the λ-calculus, and variable manipulations in substitutions are handled externally, using α-conversion and Barendregt's hygiene convention if necessary.

As usual, we introduce the classical notions of one-step, many-steps, and congruence relation of $\to_{\rho\sigma\delta}$.

Definition 4.1 [One-step, Multi-steps, Congruence of $\to_{\rho\sigma\delta}$]
Let $\mathsf{Ctx}[-]$ be any context with a single hole, and let $\mathsf{Ctx}[\mathcal{T}]$ be the result of filling the hole with the term $\mathcal{T}$;

(i) the one-step evaluation $\mapsto_{\rho\sigma\delta}$ is defined by the following inference rules:

$$\frac{\mathcal{T}_1 \to_{\rho\sigma\delta} \mathcal{T}_2}{\mathsf{Ctx}[\mathcal{T}_1] \mapsto_{\rho\sigma\delta} \mathsf{Ctx}[\mathcal{T}_2]} \quad (\mathsf{Ctx}[-])$$

where $\to_{\rho\sigma\delta}$ denotes one of the top-level rules of ρCal;

(ii) the multi-step evaluation $\mapsto\!\!\!\!\to_{\rho\sigma\delta}$ is defined as the reflexive and transitive closure of $\mapsto_{\rho\sigma\delta}$;

(iii) the congruence relation $=_{\rho\sigma\delta}$ is the symmetric closure of $\mapsto\!\!\!\!\to_{\rho\sigma\delta}$.

4.2 Rigid Pattern Condition.

When no restrictions are imposed on the shape of terms and thus on the resulting matching equations, the small-step reduction looses the confluence property. Therefore, a suitable condition should be imposed on the shape of patterns. In this paper we adopted for ρCal the so called *Rigid Pattern Condition* (RPC), that was firstly formalized in [36] and we restrict the syntax to terms of the following form:

$$\mathcal{T} ::= \mathcal{P} \twoheadrightarrow \mathcal{T} \mid [\mathcal{P} \ll \mathcal{T}]\mathcal{T} \mid \ldots \text{ as before } \ldots$$

with $\mathcal{P}$ the set of *rigid patterns* (*i.e.* terms satisfying RPC). We refer to [36,3] for a formal definition of RPC; for the purpose of this paper we just present a characterization of an "honest" subset $\mathcal{P}$ of $\mathcal{T}$ which properly contains $\mathcal{V}$ and satisfies RPC.

Definition 4.2 [Characterization of $\mathcal{P}$] Let $NF(\rho\sigma\delta)$ be the set of terms that cannot be reduced in the small-step reduction semantics; we define:

$$\mathcal{P} \triangleq \{\mathcal{T} \in NF(\rho\sigma\delta) \mid \mathcal{T} \text{ is } \textit{linear} \text{ with no } \textit{active} \text{ variables}\}$$

When a left-hand side of an application is a variable (*e.g.* in $(\mathcal{X}\ \mathcal{T})$) we say that the respective variable $\mathcal{X}$ is A *active.*

Remark 4.3 (See [36]) *The set $\mathcal{P}$ defined by the above characterization satisfies* RPC *(i.e. its elements satisfy* RPC*) and properly contains $\mathcal{V}$ (hence lambda calculus can be encoded in ρCal). $\mathcal{P}$ is not the maximal set for which $\mapsto_{\rho\sigma\delta}$ is confluent, e.g. $\mathcal{P}_\Omega \triangleq \mathcal{P} \cup \{\Omega\}$ where $\Omega \triangleq (\mathcal{X} \to \mathcal{X}\,\mathcal{X})(\mathcal{X} \to \mathcal{X}\,\mathcal{X})$ also satisfies* RPC*[36].*

Small-step reduction semantics on terms satisfying RPC is confluent.

Proposition 4.4 (Church Rosser) *The relation $\mapsto_{\rho\sigma\delta}$ is confluent.*

In order to illustrate the small-step semantics of the calculus, we present the reduction of two terms using different evaluation strategies (outermost *vs.* innermost) and yielding in the first case a "successful" result (*i.e.* containing no delayed matching constraints) and in the second one an "unsuccessful" one.

Example 4.5 [Four small-step reductions] Take the terms

$$(f(\mathcal{X}) \to (3 \to 3)\mathcal{X})\, f(3) \quad \text{and} \quad (f(\mathcal{X}) \to (3 \to 3)\mathcal{X})\, f(4)$$

We reduce them, trying four possible (underlined) redexes.

(i) $\underline{(f(\mathcal{X}) \to (3 \to 3)\mathcal{X})\, f(3)} \mapsto_\rho \underline{[f(\mathcal{X}) \ll f(3)]((3 \to 3)\,\mathcal{X})} \mapsto_\sigma \underline{(3 \to 3)\,3} \mapsto_\rho \underline{[3 \ll 3]3} \mapsto_\sigma 3$

(ii) $(f(\mathcal{X}) \to \underline{(3 \to 3)\mathcal{X}})\, f(3) \mapsto_\rho \underline{(f(\mathcal{X}) \to [3 \ll \mathcal{X}]3)\, f(3)} \mapsto_\rho \underline{[f(\mathcal{X}) \ll f(3)].([3 \ll \mathcal{X}]3)} \mapsto_\sigma \underline{[3 \ll 3]3} \mapsto_\sigma 3$

(iii) $\underline{(f(\mathcal{X}) \to (3 \to 3)\mathcal{X})\, f(4)} \mapsto_\rho \underline{[f(\mathcal{X}) \ll f(4)]((3 \to 3)\,\mathcal{X})} \mapsto_\sigma \underline{(3 \to 3)\,4} \mapsto_\rho [3 \ll 4]3$

(iv) $(f(\mathcal{X}) \to \underline{(3 \to 3)\mathcal{X}})\, f(4) \mapsto_\rho \underline{(f(\mathcal{X}) \to [3 \ll \mathcal{X}]3)\, f(4)} \mapsto_\rho \underline{[f(\mathcal{X}) \ll f(4)].([3 \ll \mathcal{X}]3)} \mapsto_\sigma [3 \ll 4]3$

It is worth noticing that the term $[3 \ll 4]3$ represents *de facto* a computation failure, which can be read as follows: "an error occurred due to a matching failure". The capability of ρCal to record failures is directly inherited from previous versions of the calculus where a special symbol null was explicitly introduced to denote computation failures.

We finish this subsection with some elaborated "object-oriented flavored" examples (see also [11]).

Example 4.6 [The Fix and $Para$ and Dna objects]

(i) Let $Fix_f \triangleq rec \to \mathcal{S} \to f(\mathcal{S}.rec)$. Then, we obtain easily $Fix_f.rec \mapsto\!\!\!\to_{\rho\sigma\delta} f(Fix_f.rec)$. This fixed point can be generalized as $Fix \triangleq rec \to \mathcal{S} \to \mathcal{X} \to \mathcal{X}(\mathcal{S}.rec(\mathcal{X}))$ and its behavior will be $Fix.rec(f) \mapsto\!\!\!\to_{\rho\sigma\delta} f(Fix.rec(f))$.

(ii) Let $Para \triangleq (par(\mathcal{X}) \to \mathcal{S} \to \mathcal{S}.\mathcal{X}, a \to \mathcal{S} \to 3, ...)$. The method $par(\mathcal{X})$ seeks for a method name that is assigned to the variable $\mathcal{X}$ and then sends (*i.e.* installs, as a first-class citizen) this method to the object itself, *i.e.*

$$\frac{}{\mathcal{V} \Downarrow \mathcal{V}} \quad (Red\text{−}Val)$$

$$\frac{\mathcal{T}_1 \Downarrow \mathcal{T}_3 \twoheadrightarrow \mathcal{T}_4 \quad [\mathcal{T}_3 \ll \mathcal{T}_2]\mathcal{T}_4 \Downarrow \mathcal{O}}{\mathcal{T}_1\, \mathcal{T}_2 \Downarrow \mathcal{O}} \quad (Red\text{−}\rho_1)$$

$$\frac{\mathcal{T}_1 \Downarrow \mathcal{T}_3, \mathcal{T}_4 \quad \mathcal{T}_3\, \mathcal{T}_2 \Downarrow \mathcal{O}_1 \quad \mathcal{T}_4\, \mathcal{T}_2 \Downarrow \mathcal{O}_2}{\mathcal{T}_1\, \mathcal{T}_2 \Downarrow \mathcal{O}_1, \mathcal{O}_2} \quad (Red\text{−}\delta)$$

$$\frac{\exists \sigma.\ \sigma(\mathcal{T}_1) \equiv \mathcal{T}_2 \quad \sigma(\mathcal{T}_3) \Downarrow \mathcal{O}}{[\mathcal{T}_1 \ll \mathcal{T}_2]\mathcal{T}_3 \Downarrow \mathcal{O}} \quad (Red\text{−}\sigma_1)$$

$$\frac{\nexists \sigma.\ \sigma(\mathcal{T}_1) \equiv \mathcal{T}_2}{[\mathcal{T}_1 \ll \mathcal{T}_2]\mathcal{T}_3 \Downarrow \mathsf{wrong}} \quad (Red\text{−}\sigma_2)$$

$$\frac{\mathcal{T}_1 \Downarrow \mathsf{wrong}}{\mathcal{T}_1\, \mathcal{T}_2 \Downarrow \mathsf{wrong}} \quad (Red\text{−}\rho_2)$$

Fig. 2. Big-step Operational Semantics

$Para.(par(a)) \triangleq Para\,(par(a))\; Para \mapsto\!\!\!\twoheadrightarrow_{\rho\sigma\delta} (\mathcal{S} \twoheadrightarrow \mathcal{S}.a)\; Para \mapsto_{\rho\sigma\delta} Para.a \mapsto\!\!\!\twoheadrightarrow_{\rho\sigma\delta} 3.$

(iii) Let $Dna \triangleq set \twoheadrightarrow \mathcal{S} \twoheadrightarrow \mathcal{X} \twoheadrightarrow ((add \twoheadrightarrow \mathcal{S}' \twoheadrightarrow \mathcal{Y} \twoheadrightarrow (\mathcal{Y}, \mathcal{S}')), \mathcal{X})$. The *set* method of *Dna* is used to create an object completely from scratch by receiving from outside all the components of a method, namely, the labels and the bodies. Once the object is installed, it has the capability to extend itself upon the reception of the message *add*. In some sense the "power" of *Dna* has been inherited by the created object. Then: $Dna.set(a \twoheadrightarrow \mathcal{S} \twoheadrightarrow 3) \triangleq Dna\,set\,Dna\,(a \twoheadrightarrow \mathcal{S} \twoheadrightarrow 3) \mapsto\!\!\!\twoheadrightarrow_{\rho\sigma\delta}$ $(\mathcal{X} \twoheadrightarrow ((add \twoheadrightarrow \mathcal{S}' \twoheadrightarrow \mathcal{Y} \twoheadrightarrow (\mathcal{Y}, \mathcal{S}')), \mathcal{X}))\,(a \twoheadrightarrow \mathcal{S} \twoheadrightarrow 3) \mapsto_{\rho\sigma\delta}$ $((add \twoheadrightarrow \mathcal{S}' \twoheadrightarrow \mathcal{Y} \twoheadrightarrow (\mathcal{Y}, \mathcal{S}')), a \twoheadrightarrow \mathcal{S} \twoheadrightarrow 3) \triangleq \mathcal{T}$, and $\mathcal{T}.add(b \twoheadrightarrow \mathcal{S} \twoheadrightarrow 4) \mapsto\!\!\!\twoheadrightarrow_{\rho\sigma\delta} \mathcal{T}, b \twoheadrightarrow \mathcal{S} \twoheadrightarrow 4.$

4.3 Big-step Operational Semantics

We define an operational semantics via a natural proof deduction system à la Plotkin [35]. The purpose of the deduction system is to map every closed expression into a normal form, *i.e.* an irreducible term in weak head normal form. The presented strategy is *lazy call-by-name* since it does not work under plain abstractions (*i.e.* $\mathcal{T} \twoheadrightarrow \mathcal{T}$), structures (*i.e.* $\mathcal{T}, \mathcal{T}$), and algebraic terms (*i.e.* $\mathcal{K}\,\mathcal{T}$). We define the set of values $\mathcal{V}$ and output values $\mathcal{O}$ as follows:

$$\mathcal{V} ::= \mathcal{K} \mid \mathcal{T} \twoheadrightarrow \mathcal{T} \mid \mathcal{K}\,\mathcal{T} \mid \mathcal{T}, \mathcal{T}$$

$$\mathcal{O} ::= \mathcal{V} \mid \mathsf{wrong} \mid \mathcal{O}, \mathcal{O}$$

The special output wrong represents the result obtained by a computation involving a "matching equation failure" (represented in [11] by null). The semantics is defined via a judgment of the shape $\mathcal{T} \Downarrow \mathcal{O}$, and its rules (almost self explaining) are presented in Figure 2.

As in the small-step reduction semantics, the big-step semantics make use of the matching algorithm $\lll_\Delta$ given in Definition 3.3. The big-step operational semantics is deterministic, and immediately suggests how to build an interpreter for the calculus. Moreover, big-step operational semantics is sound with respect to the relation $\mapsto\!\!\!\twoheadrightarrow_{\rho\sigma\delta}$, since the following holds:

Proposition 4.7 (Soundness of $\Downarrow$) *For a closed* $\mathcal{T}$*, if* $\mathcal{T} \Downarrow \mathcal{V}$*, then* $\mathcal{T} \mapsto\!\!\!\twoheadrightarrow_{\rho\sigma\delta} \mathcal{V}$.

Definition 4.8 [Convergence of $\Downarrow$] For a closed $\mathcal{T}$, we say that $\mathcal{T}$ converges, and we write it as $\mathcal{T} \Downarrow$, if there exists an output value $\mathcal{O}$, such that $\mathcal{T} \Downarrow \mathcal{O}$.

Given the above definition, we also conjecture the completeness, which shows that every terminating program also terminates in our interpreter.

Proposition 4.9 (Completeness of $\Downarrow$) *For a closed* $\mathcal{T}$*, if* $\mathcal{T} \mapsto\!\!\!\twoheadrightarrow_{\rho\sigma\delta} \mathcal{V}$*, then* $\mathcal{T}$ *converges, i.e.* $\mathcal{T} \Downarrow$.

Exercise. [Call-by-value] If you want to play with big-step operational semantics, then modify the set of output values and the deduction rules of $\Downarrow$ in order to implement a *lazy call-by-value* strategy.

In order to illustrate the behavior of big-step semantics, we present the deduction trees of the previous two terms.

Example 4.10 [Two big-step derivations] Take the terms

$$(f(\mathcal{X}) \rightarrow (3 \rightarrow 3)\mathcal{X})\, f(3) \quad \text{and} \quad (f(\mathcal{X}) \rightarrow (3 \rightarrow 3)\mathcal{X})\, f(4)$$

The deduction trees are:

$$\dfrac{\dfrac{}{*} \quad \dfrac{\sigma \equiv \{3/\mathcal{X}\} \quad \begin{matrix}\vdots \\ (3 \rightarrow 3)3 \Downarrow 3\end{matrix}}{[f(\mathcal{X}) \ll f(3)](3 \rightarrow 3)\mathcal{X} \Downarrow 3}}{(f(\mathcal{X}) \rightarrow (3 \rightarrow 3)\mathcal{X})\, f(3) \Downarrow 3} \quad \text{and} \quad \dfrac{\dfrac{}{*} \quad \dfrac{\sigma \equiv \{4/\mathcal{X}\} \quad \dfrac{\nexists \sigma.\ \sigma(3) \equiv 4}{(3 \rightarrow 3)4 \Downarrow \mathsf{wrong}}}{[f(\mathcal{X}) \ll f(4)](3 \rightarrow 3)\mathcal{X} \Downarrow \mathsf{wrong}}}{(f(\mathcal{X}) \rightarrow (3 \rightarrow 3)\mathcal{X})\, f(4) \Downarrow \mathsf{wrong}}$$

with $* \equiv (f(\mathcal{X}) \rightarrow (3 \rightarrow 3)\mathcal{X}) \Downarrow (f(\mathcal{X}) \rightarrow (3 \rightarrow 3)\mathcal{X})$.

In the big-step operational semantics, the output value wrong is used in order to denote "bad" computations where matching failure occurs at run-time. As such, the presented semantics does not abort computations, as for example in

$$\dfrac{\vdots}{(3 \rightarrow 3, 4 \rightarrow 4)\, 4 \Downarrow \mathsf{wrong}, 4}$$

keeping in the final result the fact that one computation goes wrong. Of course, other choices are possible, like the one of "killing" the computation once a wrong value is produced, as in

$$\frac{\vdots}{(3 \twoheadrightarrow 3, 4 \twoheadrightarrow 4)\, 4 \Downarrow \mathsf{wrong}}$$

This latter choice would need the modification of our big-step semantics. To resume: the first (presented) choice leads to an "optimistic" machine (at least one computation does not go wrong), while the latter choice leads to a "pessimistic" one (the machine stops if at least one wrong occurs).

Exercise. If you love natural semantics:

(i) (pessimistic machine) modify the presented "optimistic" big-step semantics into a "pessimistic" one, *e.g.* add/modify all deduction rules raising (in premises) and propagating (from premises to conclusion) the wrong value;

(ii) (example 4.6) reduce, using the big-step operational semantics, the *Para* and *Dna* objets: check whether the same results as for the small-step reduction semantics are obtained.

5 Dealing with Exceptions

In the previous section, we saw that the wrong output value aborts computations. Nevertheless, in modern programming languages one may be interested in trying some chunks of code in a protected environment, and whenever a run-time matching error occurs, catch it first, and then execute some exception handler, which can be declared many "miles" away from where the exception was raised. This is the case of the `try{...}catch{...}` mechanism of Java, or similar constructs in ML, $C^{\#}$, ... In ρCal a possible exception can signal that some matching failure occurred at run-time, such as matching the constant 3 against another constant 4. In this case the term $(3 \twoheadrightarrow 3)\, 4$ reduces to $[3 \ll 4]3$ which can be read as follows:

> *"the result would be* 3 *but at run-time the program tried to match* 3 *against* 4*, yielding the (dirty) result* $[3 \ll 4]3$*"*

In the following, we present a comfortable extension of ρCal which takes into account matching exceptions and their handling. A simple exception handling mechanism for the rewriting-calculus has been already studied in [16]. By lack of space we present here only the big-step operational semantics, leaving the small-step one as an exercise. This extension was inspired from [28]. To do this we need first to add an exception handling constructor to the ρCal syntax, *i.e.*

$\mathcal{T} ::= \mathsf{try}\ \mathcal{T}\ \mathsf{catch}\ [\mathcal{T} \ll \mathcal{T}]\ \mathsf{with}\ \mathcal{T} \mid$ ... as before ...

Intuitively, $[\mathcal{T} \ll \mathcal{T}]$ denotes a matching equation without solutions, like *e.g.* $[3 \ll 4]$. Executing a $\mathsf{try}\ \mathcal{T}_1\ \mathsf{catch}\ [\mathcal{T}_2 \ll \mathcal{T}_3]\ \mathsf{with}\ \mathcal{T}_4$ means that the scope

$(Red{-}Val), (Red{-}\rho_1), (Red{-}\sigma_1)$ as in Figure 2

$$\frac{\nexists \sigma.\ \sigma(\mathcal{T}_1) \equiv \mathcal{T}_2}{[\mathcal{T}_1 \ll \mathcal{T}_2]\mathcal{T}_3 \Downarrow [\mathcal{T}_1 \ll \mathcal{T}_2]} \quad (Red{-}\sigma_2)$$

$$\frac{\mathcal{T}_1 \Downarrow [\mathcal{T}_2 \ll \mathcal{T}_3] \quad \mathcal{T}_4 \Downarrow \mathcal{O}}{\mathsf{try}\ \mathcal{T}_1\ \mathsf{catch}\ [\mathcal{T}_2 \ll \mathcal{T}_3]\ \mathsf{with}\ \mathcal{T}_4 \Downarrow \mathcal{O}} \quad (Red{-}Exc_1)$$

$$\frac{\mathcal{T}_1 \Downarrow \mathcal{O} \quad \mathcal{O} \not\equiv [\mathcal{T}_2 \ll \mathcal{T}_3]}{\mathsf{try}\ \mathcal{T}_1\ \mathsf{catch}\ [\mathcal{T}_2 \ll \mathcal{T}_3]\ \mathsf{with}\ \mathcal{T}_4 \Downarrow \mathcal{O}} \quad (Red{-}Exc_2)$$

$$\frac{\mathcal{T}_1 \Downarrow \mathcal{T}_3, \mathcal{T}_4 \quad \mathcal{T}_3\,\mathcal{T}_2 \Downarrow \mathcal{V}_1 \quad \mathcal{T}_4\,\mathcal{T}_2 \Downarrow \mathcal{V}_2}{\mathcal{T}_1\,\mathcal{T}_2 \Downarrow \mathcal{V}_1, \mathcal{V}_2} \quad (Red{-}\delta)$$

$$\frac{\mathcal{T}_1 \Downarrow [\mathcal{T}_3 \ll \mathcal{T}_4]}{\mathcal{T}_1\,\mathcal{T}_2 \Downarrow [\mathcal{T}_3 \ll \mathcal{T}_4]} \quad (Red{-}Prop_1)$$

$$\frac{\mathcal{T}_1 \Downarrow \mathcal{T}_3, \mathcal{T}_4 \quad \mathcal{T}_3\,\mathcal{T}_2 \Downarrow [\mathcal{T}_5 \ll \mathcal{T}_6]}{\mathcal{T}_1\,\mathcal{T}_2 \Downarrow [\mathcal{T}_5 \ll \mathcal{T}_6]} \quad (Red{-}Prop_2)$$

$$\frac{\mathcal{T}_1 \Downarrow \mathcal{T}_3, \mathcal{T}_4 \quad \mathcal{T}_3\,\mathcal{T}_2 \Downarrow \mathcal{V} \quad \mathcal{T}_4\,\mathcal{T}_2 \Downarrow [\mathcal{T}_5 \ll \mathcal{T}_6]}{\mathcal{T}_1\,\mathcal{T}_2 \Downarrow [\mathcal{T}_5 \ll \mathcal{T}_6]} \quad (Red{-}Prop_3)$$

Fig. 3. Big-step dealing with Exception Handling

of the matching failure $[\mathcal{T}_2 \ll \mathcal{T}_3]$ is active in $\mathcal{T}_1$, and that if that exception occurs, the handler $\mathcal{T}_4$ will be executed. Otherwise the raised exception will be propagated outside the scope of the try_catch_with.

More precisely: first, we evaluate $\mathcal{T}_1$ (the protected term). If $\mathcal{T}_1$ evaluates to an output without matching failures (*i.e.* an output value), then this value will be the result of the whole try_catch_with expression. This roughly corresponds to executing $\mathcal{T}_1$ without raising exceptions.

Counterwise, if a matching failure occurs, then we must check whether the failure is the one declared in the try_catch_with expression (*i.e.* $[\mathcal{T}_2 \ll \mathcal{T}_3]$) or not; in the first case we execute the handler $\mathcal{T}_4$, while in the latter case we propagate the matching failure outside the scope of the try_catch_with expression. Note that the scope of the exception $[\mathcal{T}_2 \ll \mathcal{T}_3]$ ranges over $\mathcal{T}_1$ but not $\mathcal{T}_4$. In order to add an exception handling mechanism, we first modify the set of output values $\mathcal{O}$ as follows:

$$\mathcal{V} ::= \mathcal{K} \mid \mathcal{T} \twoheadrightarrow \mathcal{T} \mid \mathcal{K}\,\mathcal{T} \mid \mathcal{T}, \mathcal{T}$$

$$\mathcal{O} ::= \mathcal{V} \mid [\mathcal{T} \ll \mathcal{T}]$$

Intuitively, instead of introducing a simple wrong output signal, the special output $[\mathcal{T} \ll \mathcal{T}]$ customizes exception signals (hence records the kind of matching failure) and ensures that exceptions can be propagated and caught correctly. Therefore, we keep the rules $(Red{-}Val)$, $(Red{-}\rho_1)$, and $(Red{-}\sigma_1)$ and we add some extra deduction rules as shown in Figure 3.

The presented interpreter implements a "pessimistic" machine that strictly propagates the exception signals. Thus, one should notice that, according to the last three (propagation) rules, an exception signal is propagated independently of the other (possibly successful) computation. Note that the $(Red{-}Exc_2)$ can also be seen as a propagation rule since an exception signal different from the one specified in the try_catch_with expression is strictly propagated. If no matching failure is generated in the protected part, then the corresponding result is propagated.

Example 5.1 [One big-step derivation] The term

$$\mathsf{try}\ (f(\mathcal{X}) \rightarrowtriangle (3 \rightarrowtriangle 3)\mathcal{X})\, f(4)\ \mathsf{catch}\ [3 \ll 4]\ \mathsf{with}\ g(4)$$

is evaluated using the natural deduction showed below,

$$\dfrac{\dfrac{\dfrac{}{*} \qquad \dfrac{\sigma \equiv \{4/\mathcal{X}\} \quad \dfrac{\nexists \sigma.\ \sigma(3) \equiv 4}{(3 \rightarrowtriangle 3)4 \Downarrow [3 \ll 4]}}{[f(\mathcal{X}) \ll f(4)](3 \rightarrowtriangle 3)\mathcal{X} \Downarrow [3 \ll 4]}}{(f(\mathcal{X}) \rightarrowtriangle (3 \rightarrowtriangle 3)\mathcal{X})\, f(4) \Downarrow [3 \ll 4]} \qquad \dfrac{}{f(4) \Downarrow f(4)}}{\mathsf{try}\ (f(\mathcal{X}) \rightarrowtriangle (3 \rightarrowtriangle 3)\mathcal{X})\, f(4)\ \mathsf{catch}\ [3 \ll 4]\ \mathsf{with} f(4) \Downarrow f(4)}$$

with $* \equiv (f(\mathcal{X}) \rightarrowtriangle (3 \rightarrowtriangle 3)\mathcal{X}) \Downarrow (f(\mathcal{X}) \rightarrowtriangle (3 \rightarrowtriangle 3)\mathcal{X})$.

Exercise. (i) (small-step semantics) Design the small-step reduction semantics for the ρCal enhanced with the new try_catch_with exception handler (hint: check the $\mathcal{C}$ and $\mathcal{A}$ control operators of M. Felleisen [17]);

(ii) (generalized exceptions) refine and generalize $\Downarrow$ in order to catch generalized failures like $[\mathcal{T}_1 \ll \mathcal{T}_2]$ representing any unsolvable matching equation trying to match a *fixed* term $\mathcal{T}_1$ against *any* term $\mathcal{T}_3$, such that $\mathcal{T}_2 \ll \mathcal{T}_3$ is solvable;

(iii) (more generalization) refine and generalize $\Downarrow$ of (ii) in order to declare generalized failures like $[\mathcal{T}_1 \ll \mathcal{T}_2]$ and capture any exception like $[\mathcal{T}_3 \ll \mathcal{T}_4]$, such that $\mathcal{T}_1 \ll \mathcal{T}_3$ and $\mathcal{T}_2 \ll \mathcal{T}_4$ are solvable;

(iv) (call-by-value) as in the previous exercise, modify $\Downarrow$ in the extended ρCal with exceptions in order to implement a *lazy call-by-value* strategy.

6 Polymorphic Type Inference

We have presented so far an untyped (à la Curry) syntax where the terms are not decorated with types. This section presents a simple type discipline which can be used to assign a semantical meaning to ρCal programs by statically type-checking and hence, catching some errors (unfortunately not the run-time matching one) before running the ρCal-terms.

Sophisticated typed systems were presented for the Rewriting Calculus and for lambda calculi with pattern facilities in [12,3]; those type disciplines range over simple, polymorphic, dependent, and higher-order types, following the "cubism" folklore of H. Barendregt [2].

Different type systems correspond in practice to different utilization of the ρCal: for example, dependent and higher-order types are widely used in theorem provers based on the Curry-Howard isomorphism, like Coq or Isabelle, while polymorphic types are mostly used to implement type inference algorithms for (functional) programming languages with pattern matching facilities, in the style of ML.

Here, we focus on the polymorphic type discipline; we do not discuss here the decidability of this system which is essentially an untyped presentation of the J.Y. Girard's system F [20], as presented in D. Leivant's polymorphic lambda calculus [27]. Nevertheless, we conjecture that the classical restrictions on universal quantifiers, as the one adopted in ML [13], suitably customized with pattern facilities, could apply also for our ρCal.

The syntax of types and contexts is defined as follows (σ, τ range over types, and α ranges over type-variables, and Γ ranges over contexts):

$$\sigma ::= \alpha \mid \forall \alpha.\sigma \mid \sigma \twoheadrightarrow \sigma$$
$$\Gamma ::= \emptyset \mid \Gamma, \mathcal{X}{:}\sigma \mid \Gamma, \mathcal{K}{:}\sigma \mid \alpha{:}\sigma$$

A type judgment in ρCal has the shape $\Gamma \vdash \mathcal{T} : \sigma$. The type inference system is defined by the rule schemas presented in Figure 4. In what follows, we briefly give a guided tour of the type inference rules and insist on the most intriguing ones.

- The (*Start*) rules are standard and need no comments.
- The (*Struct*) rule says that a structure $(\mathcal{T}_1, \mathcal{T}_2)$, can be typed with the type σ, provided that the same σ can be assigned to $\mathcal{T}_1$ and $\mathcal{T}_2$.
- The (*Abs*) rule deals with abstractions in which we bind over (non trivial) patterns instead of variables; note that the context Γ' is charged in the premises in order to take into account the type of the free variables of $\mathcal{T}_1$ (possibly bound in $\mathcal{T}_2$).
- The (*Match*) rule deals with terms in which a *delayed matching equation* occurs *hard-coded* into the term; this rule is essentially needed to ensure the well-typedness of terms leading to matching failures (*e.g.* $(3 \twoheadrightarrow 3)\,\mathcal{X}$) and ensure the subject reduction property for the top-level rules (ρ) and (σ). Again, Γ' records the type of the free variables of $\mathcal{T}_1$ (possibly bound in $\mathcal{T}_3$).

$$\frac{\mathcal{X}{:}\sigma \in \Gamma}{\Gamma \vdash \mathcal{X} : \sigma}\ (Start_1) \qquad \frac{\mathcal{K}{:}\sigma \in \Gamma}{\Gamma \vdash \mathcal{K} : \sigma}\ (Start_2)$$

$$\frac{\Gamma \vdash \mathcal{T}_1 : \sigma \quad \Gamma \vdash \mathcal{T}_2 : \sigma}{\Gamma \vdash \mathcal{T}_1, \mathcal{T}_2 : \sigma}\ (Struct)$$

$$\frac{\Gamma, \Gamma' \vdash \mathcal{T}_1 : \sigma \quad \Gamma, \Gamma' \vdash \mathcal{T}_2 : \tau \quad Dom(\Gamma') = FV(\mathcal{T}_1)}{\Gamma \vdash \mathcal{T}_1 \to \mathcal{T}_2 : \sigma \to \tau}\ (Abs)$$

$$\frac{\Gamma, \Gamma' \vdash \mathcal{T}_1 : \sigma \quad \Gamma \vdash \mathcal{T}_2 : \sigma \quad \Gamma, \Gamma' \vdash \mathcal{T}_3 : \tau \quad Dom(\Gamma') = FV(\mathcal{T}_1)}{\Gamma \vdash [\mathcal{T}_1 \ll \mathcal{T}_2]\mathcal{T}_3 : \tau}\ (Match)$$

$$\frac{\Gamma \vdash \mathcal{T}_1 : \sigma \to \tau \quad \Gamma \vdash \mathcal{T}_2 : \sigma}{\Gamma \vdash \mathcal{T}_1\ \mathcal{T}_2 : \tau}\ (Appl)$$

$$\frac{\Gamma \vdash \mathcal{T} : \sigma \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash \mathcal{T} : \forall\alpha.\sigma}\ (Abs{-}\forall)$$

$$\frac{\Gamma \vdash \mathcal{T} : \forall\alpha.\sigma}{\Gamma \vdash \mathcal{T} : \sigma\{\tau/\alpha\}}\ (Appl{-}\forall)$$

Fig. 4. Polymorphic Type Inference

- The (*Appl*) rule is standard and needs no comments.
- The (*Abs*$-\forall$) and the (*Appl*$-\forall$) rules introduce (resp. eliminate) polymorphic types: they are standard as in Leivant's polymorphic lambda calculus. Note that those two rules are not syntax directed.

Theorem 6.1 (Subject Reduction) *If* $\Gamma \vdash \mathcal{T}_1 : \sigma$ *and* $\mathcal{T}_1 \mapsto\!\!\!\twoheadrightarrow_{\rho\delta} \mathcal{T}_2$, *then* $\Gamma \vdash \mathcal{T}_2 : \sigma$.

Exercise. If you really love type systems:

(i) (normalization) show that all typable terms are strongly normalizing;

(ii) (feasable inference) find a suitable extension of the algorithm W of Damas-Milner [13] which fits with ρCal; study the complexity;

(iii) (typing exceptions) customise the type system in order to take into account try_catch_with exceptions (hint: check some works on exceptions in ML [21,33]);

(iv) (challenge) using the pessimistic big-step machine, show the (un)decidability of the following type soundness proposition: $\emptyset \vdash \mathcal{T} : \sigma$, and $\mathcal{T} \Downarrow \mathcal{O}$, then $\mathcal{O} \neq \mathsf{wrong}$.

7 Conclusions

With this little "pilgrimage" we hope to have contributed to the understanding of some basic concepts of the Rewriting Calculus which is a relatively young (but powerful) formalism. The calculus provides a well-behaved integration of (higher order) term rewriting systems and lambda calculus.

The calculus is suitable to further extensions and improvements. The possibility to plug-in sophisticated type theories open the road for the study of new powerful proof engines and (meta)languages.

New conditions less restrictive than RPC that would allow one a larger class of patterns in abstractions are worth studying.

Conceiving a denotational semantics with continuations dealing with exceptions and sophisticated (user customizable) strategies is also worth studying, the final aim being the integration of functional, and logic, and rule based programming paradigms.

The challenge of building a new type system which statically prevents the run-time `match-fail` errors (although we conjecture the undecidability) is very stimulating.

Exploring a limited form of decidable higher-order unification, in the style of λ-Prolog [29,30] is also challenging, the goal being to improve the automatization of theorem provers.

Finally, we conjecture that a suitable theory would allow one to deal with concurrency and, hopefully, with mobility, in the style of Join Calculus [18], Ambients [5], or Mobile Maude [14].

Acknowledgments.

Luigi would like to thank Furio Honsell and Simonetta Ronchi della Rocca, for the time spent to teach him the main "tips & tricks" underneath ML/Scheme exceptions and Girard's system F. We all thank Benjamin Wack for a careful reading of the manuscript.

References

[1] H. Barendregt. *Lambda Calculus: its Syntax and Semantics.* North Holland, 1984.

[2] H. Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, volume II, pages 118–310. Oxford University Press, 1992.

[3] G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure Pattern Type Systems. In The ACM press, editor, *Proc. of POPL*, 2003.

[4] V. Breazu-Tannen. Combining Algebra and Higher-order Types. In *Proc. of LICS*, pages 82–90, 1988.

[5] L. Cardelli and A. D. Gordon. Mobile Ambients. *Theoretical Computer Science*, 240(1), 2000.

[6] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1941.

[7] H. Cirstea. *Calcul de Réécriture : Fondements et Applications.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, 2000.

[8] H. Cirstea and C. Kirchner. ρ-calculus. Its Syntax and Basic Properties. In *Proc. of Workshop CCL*, 1998.

[9] H. Cirstea and C. Kirchner. An Introduction to the Rewriting Calculus. Research Report RR-3818, INRIA, 1999.

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

[11] H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In *Proc. of RTA*, volume 2051 of *LNCS*, pages 77–92. Springer-Verlag, 2001.

[12] H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In *Proc. of FOSSACS*, volume 2030 of *LNCS*, pages 166–180, 2001.

[13] L. Damas and R. Milner. Principal Type-Schemes for Functional Programs. In *Proc. of POPL*, pages 207–212. The ACM Press, 1982.

[14] F. Durán, S. Eker, P. Lincoln, and J. Meseguer. Principles of Mobile Maude. In *Proc. of ASA/MA*, volume 4 of *LNCS*, pages 73–85. Springer-Verlag, 2000.

[15] S. Eker. Associative-Commutative Matching Via Bipartite Graph Matching. *The Computer Journal*, 38(5):381–399, 1995.

[16] G. Faure and C. Kirchner. Exceptions in the Rewriting Calculus. In *Proc. of RTA*, volume 2378 of *LNCS*, pages 66–82. Springer-Verlag, 2002.

[17] M. Felleisen and D. P. Friedman. A Syntactic Theory of Sequential State. *Theoretical Computer Science*, 69:243–287, 1989.

[18] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A Calculus of Mobile Agents. In *Proc. of CONCUR*, volume 1119 of *LNCS*, pages 406–421. Springer-Verlag, 1996.

[19] J. Gallier and V. Breazu-Tannen. Polymorphic Rewriting Conserves Algebraic Strong Normalization and Confluence. In *Proc. of ICALP*, volume 372 of *LNCS*, pages 137–150. Springer-Verlag, 1989.

[20] J.Y. Girard. The System F of Variable Types, Fifteen Years Later. *Theoretical Computer Science*, 45:159–192, 1986.

[21] C. A. Gunter, D. Rémy, and J. G. Riecke. A Generalization of Exceptions and Control in ML-like Languages. In *Proc. of FPCA*, volume 2378, pages 66–82. The ACM Press, 1995.

[22] J.-M. Hullot. Associative-Commutative Pattern Matching. In *Proc. of IJCAI*, 1979.

[23] J.P. Jouannaud and M. Okada. Abstract Data Type Systems. *Theoretical Computer Science*, 173(2):349–391, 1997.

[24] S. N. Kamin. Inheritance in Smalltalk-80: A Denotational Definition. In The ACM press, editor, *Proc. of POPL*, pages 80–87, 1988.

[25] D. Kesner, L. Puel, and V. Tannen. A Typed Pattern Calculus. *Information and Computation*, 124(1):32–61, 10 1996.

[26] J.W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory Reduction Systems: Introduction and Survey. *Theoretical Computer Science*, 121:279–308, 1993. Special issue in honour of Corrado Böhm.

[27] D. Leivant. Polymorphic Type Inference. In *Proc. of POPL*, pages 88–98. The ACM press, 1983.

[28] L. Liquori. Semantica e Pragmatica di un Linguaggio Funzionale con le Continuazioni Esplicite. Laurea in Science dell'Informazione, University of Udine, 1990. In Italian, 74 pp.

[29] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In *Proc. of ELP*, volume 475 of *LNCS*, pages 253–281. Springer-Verlag, 1991.

[30] D. Miller, G. Nadathur, F. Pfenning, and A. Shedrov. Uniform Proofs as a Foundation for Logic Programming. *Annals of Pure and Applied Logics*, 51:125–157, 1991.

[31] T. Nipkow and C. Prehofer. Higher-Order Rewriting and Equational Reasoning. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications. Volume I: Foundations.* Kluwer, 1998.

[32] M. Okada. Strong Normalizability for the Combined System of the Typed λ Calculus and an Arbitrary Convergent Term Rewrite System. In *Proc. of ISSAC*, pages 357–363. ACM Press, 1989.

[33] F. Pessaux and X. Leroy. Type-based Analysis of Uncaught Exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, 2000.

[34] S. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[35] G. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, 1981.

[36] V. van Oostrom. Lambda Calculus with Patterns. Technical Report IR-228, Faculteit der Wiskunde en Informatica, Vrije Universiteit Amsterdam, 1990.

[37] D. A. Wolfram. *The Clausal Theory of Types*, volume 21 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1993.