Rewriting Calculus with Fixpoints: Untyped and First-order Systems

Horatiu Cirstea, Luigi Liquori, and Benjamin Wack

LORIA & NANCY II & INRIA & NANCY I
54506 Vandoeuvre-lès-Nancy, BP 239 Cedex France
[First.Last]@loria.fr

Abstract. The rewriting calculus, also called ρ-calculus, is a framework embedding λ-calculus and rewriting capabilities, 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. Many type systems for the λ-calculus can be generalized to the ρ-calculus: in this paper, we study extensively a first-order ρ-calculus *à la* Church, called $\rho_{\rightarrow}^{\mathsf{stk}}$.
The type system of $\rho_{\rightarrow}^{\mathsf{stk}}$ allows one to type (object oriented flavored) fixpoints, leading to an expressive and safe calculus. In particular, using pattern matching, one can encode and typecheck term rewriting systems in a natural and automatic way. Therefore, we can see our framework as a starting point for the theoretical basis of a powerful typed rewriting-based language.

Keywords. Rewriting-calculus, Lambda-calculus, Object-calculus, Pattern Matching, Fixpoints, Type Theory.

1 Introduction

It is not by chance that pattern matching appears as the core mechanism of term rewriting: in fact, the ability to discriminate patterns is present since the beginning of information processing modeling. Pattern matching has also been widely used in functional programming (*e.g.* ML, Haskell, Scheme), logic programming (*e.g.* Prolog), rewrite based programming (*e.g.* Elan [5], Maude [16], script programming (*e.g.* sed, awk). It has been generally considered as a convenient mechanism for expressing complex requirements about the argument of a function, more than a real computation paradigm.

The *Rewriting Calculus*, by unifying λ-calculus and 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 rewrite rule becomes a first-class object, which can be created, manipulated and modified by the calculus itself. We have already shown [8] that the first version of the rewriting calculus can be used as an operational semantics for rewriting based languages and in particular for Elan. For this we have used in the past fixpoint operators inspired from the ones of the λ-calculus and thus untypable in the early version of the simply typed rewriting calculus [7].

Nevertheless, static analysis via a suitable typing system enforces a stronger programming discipline. The main objective of this paper is to present a ρ-calculus *à la* Church ($\rho_{\rightarrow}^{\mathsf{stk}}$) featuring first-order types and well-typed self-duplicating terms.

In $\rho_{\rightharpoondown}^{\mathsf{stk}}$ (typed) pattern matching is the basic mechanism for programming allowing one to build and typecheck non-normalizing terms: this enables the definition of some interesting *functional recursion operators*. Moreover, the type system of $\rho_{\rightharpoondown}^{\mathsf{stk}}$ is powerful enough to ensure well-typedness of matching equations, *i.e*. the instantiation of formal parameters complies with the typing discipline. Hence, $\rho_{\rightharpoondown}^{\mathsf{stk}}$ represents a good trade-off between the flexibility and the expressiveness of the untyped calculus, and the strictness of a more strongly typed one. This leads us to consider the presented typed system for ρ-calculus as a good candidate for giving the static semantics of a family of rewriting-based languages such as Elan, Maude, *etc*.

One of the particularities of the type system of $\rho_{\rightharpoondown}^{\mathsf{stk}}$ is that it relaxes, using the well-known result of N.P. Mendler [15], the classical property that "well-typed programs normalize". More precisely, non-termination can be type-checked in $\rho_{\rightharpoondown}^{\mathsf{stk}}$ thanks to *ad hoc* patterns; it follows that, roughly speaking, an ML-like let becomes a let rec by abstracting over a suitable algebraic pattern P.

Nevertheless, it is important to remark that when the type discipline is enhanced with dependent types, as it was done recently by the authors [3], all the programs presented in this paper are statically rejected, *i.e*. blocked by the type system. The chosen dependent type theory introduces pattern matching inside types, and matching failures significantly restrict the set of type-checked programs. In fact, the present paper does not fit into the philosophy of [3], where (dependent) type systems were studied especially for logical (proof-oriented) purposes and thus concerned with strong normalization of typable terms.

Plan of the paper. In Section 2, we will describe the syntax and the evaluation rules of the $\rho_{\rightharpoondown}^{\mathsf{stk}}$. We will see how an equivalence on terms handles the undesirable matching failures. Section 3 describes the type system of $\rho_{\rightharpoondown}^{\mathsf{stk}}$. We give some simple type derivations to show how the type system deals with patterns and we state metatheoretical properties of $\rho_{\rightharpoondown}^{\mathsf{stk}}$. In Section 4, we explain how a careful use of the pattern matching capabilities allows us to encode various object calculi and term rewriting systems.

2 The System $\rho_{\rightharpoondown}^{\mathsf{stk}}$

This section presents the basis of $\rho_{\rightharpoondown}^{\mathsf{stk}}$: its syntax, its semantics and some examples showing its expressiveness.

2.1 Syntax

In this paper, we consider the meta-symbols "$\rightharpoondown$" (function- and type-abstraction), "$[\ll]$" (delayed matching constraint), an application operator denoted by concatenation, and "," (structure operator). We assume that the application operator associates to the left, while the other operators associate to the right. The priority of the application is higher than that of "$[\ll]$" which is higher than that of "$\rightharpoondown$ " which is, in turn, of higher priority than the ",". The symbol τ ranges over the set $\mathcal{T}y$ of types, the symbol ι ranges over the set $\mathcal{K}y$ of type constants ($\mathcal{K}y \subseteq \mathcal{T}y$), the symbols $A, B, C, \ldots$ range over the set $\mathcal{T}$ of terms, the symbols $X, Y, Z, \ldots$ range over the set $\mathcal{V}$ of variables ($\mathcal{V} \subseteq \mathcal{T}$), the symbols $a, b, c, \ldots, f, g, h$ range over a set $\mathcal{K}$ of term constants ($\mathcal{K} \subseteq \mathcal{T}$). Finally, the symbols P, Q range over the set $\mathcal{P}$ of patterns, ($\mathcal{V} \subseteq \mathcal{P} \subseteq \mathcal{T}$). Sometimes we will use the "overloaded" symbol $\alpha \in \mathcal{V} \cup \mathcal{K}$, and we denote $\overline{A}$ for $A_1 \cdots A_n$, for $n \geq 0$. The syntax is presented in Figure 1. The *types* are as one would expect from a first-order

$\tau ::= \iota \mid \tau \twoheadrightarrow \tau$		$\mathcal{T}y$	Types
$\Delta ::= \emptyset \mid \Delta, X{:}\tau \mid \Delta, f{:}\tau$			Contexts
$P ::= X \mid \mathsf{stk} \mid f\,\overline{P}$	*(variables occur only once in any P)*	$\mathcal{P}$	Patterns
$A ::= f \mid \mathsf{stk} \mid X \mid P \twoheadrightarrow_{\Delta} A \mid [P \ll_{\Delta} A]A \mid A\;A \mid A, A$		$\mathcal{T}$	Terms

Fig. 1. Syntax of $\rho_{\twoheadrightarrow}^{\mathsf{stk}}$

type system, *i.e*. constant-types and arrow-types. The *patterns* are algebraic terms (*i.e*. terms constructed only with variables, constants and application) which can be used as left-hand sides of the rewrite rules; the set of patterns is obviously included in the set of terms. The well-known linearity restriction [17] is needed to keep the small-step semantics confluent. A *rewrite rule* of the form $(P \twoheadrightarrow_{\Delta} A)$ abstracting over the free variables of P is a first-class citizen of the calculus. The types of the free variables of P are declared in Δ, *i.e*. $\mathsf{Fv}(P) = \mathsf{Dom}(\Delta)$, resulting in a fully annotated calculus *à la* Church. An *application* is implicitly denoted by concatenation. The *delayed matching constraint* $[P \ll_{\Delta} A]B$ can be seen as the term B with its free variables constrained by the matching between P and A. Again, the context Δ contains the type declarations of all the free variables appearing in the pattern P. A *structure* is a collection of terms that can be seen either as a set of rewrite rules or as a set of results. As we will see in Section 2.3, the symbol stk can be considered as the special constant representing a delayed matching constraint whose matching problem is unsolvable. An alternative approach would be to omit this symbol from the syntax but this has two drawbacks: first, the axioms of the theory presented in Def. 5 would become more complicated; moreover, we would lose the expression of first given in Section 4.2, and thus the proposed encoding of rewriting systems would be no longer possible.

A type judgment (defined in Section 3) stating that a term A has the type τ in a context Γ is written $\Gamma \vdash A : \tau$.

Free Variables and Substitutions. We introduce the notion of free variable and substitution.

Definition 1 (Free variables Fv).

$$
\begin{array}{lllll}
\mathsf{Fv}(f) & \triangleq \emptyset & \mathsf{Fv}(P \twoheadrightarrow_{\Delta} A) & \triangleq & \mathsf{Fv}(A) \setminus \mathsf{Fv}(P) \\
\mathsf{Fv}(\mathsf{stk}) & \triangleq \emptyset & \mathsf{Fv}([P \ll_{\Delta} A]B) & \triangleq & \mathsf{Fv}((P \twoheadrightarrow_{\Delta} B)\ A) \\
\mathsf{Fv}(X) & \triangleq \{X\} & \mathsf{Fv}(A\,B) \triangleq \mathsf{Fv}(A, B) & \triangleq & \mathsf{Fv}(A) \cup \mathsf{Fv}(B)
\end{array}
$$

As usual, we work modulo α*-conversion* and we adopt Barendregt's *"hygiene-convention"* [2], *i.e.* free and bound variables have different names. This allows us to define substitutions quite straightforwardly, since it avoids issues like variable capture.

Definition 2 (Substitutions).
A substitution θ is a mapping from the set of variables to the set of terms. A finite substitution θ has the form $\{A_1/X_1 \ldots A_m/X_m\}$, and its domain $\{X_1, \ldots, X_m\}$ is denoted by $\mathsf{Dom}(\theta)$. The application of a substitution θ to a term A, denoted by $A\theta$, is defined as follows:

$$
\begin{array}{lll}
f\theta \triangleq f & (P \twoheadrightarrow_{\Delta} A)\theta & \triangleq P \twoheadrightarrow_{\Delta} A\theta \\
\mathsf{stk}\theta \triangleq \mathsf{stk} & ([P \ll_{\Delta} A]B)\theta & \triangleq [P \ll_{\Delta} A\theta]B\theta \\
X_i\theta \triangleq \begin{cases} A_i & \mathsf{if}\, X_i \in \mathsf{Dom}(\theta) \\ X_i & \mathsf{otherwise} \end{cases} & (A\,B)\theta & \triangleq A\theta\,B\theta \\
 & (A, B)\theta & \triangleq A\theta, B\theta
\end{array}
$$

A substitution θ is well-typed in context Γ *if for any $X \in \mathsf{Dom}(\theta)$ such that $\Gamma \vdash X : \tau$ we have $\Gamma \vdash X\theta : \tau$.*

Matching Equations, Theories. The core mechanism of the rewriting calculus is pattern matching since, as we have already mentioned, when a delayed matching constraint is evaluated the corresponding matching problem should be solved. We define first the classical notions of matching equations and matching solutions.

Definition 3 (Matching).
Given a theory $\mathbb{T}$ (i.e. a set of axioms defining a congruence relation $\stackrel{\mathbb{T}}{=}$):

1. *A matching equation is a problem $\mathsf{T} \triangleq P \prec\!\!\prec_{\mathbb{T}} A$ with P a pattern and A a term.*
2. *A substitution θ is a solution of the matching equation T if:*
   (a) $P\theta \stackrel{\mathbb{T}}{=} A$
   (b) θ is well-typed in any context Γ in which A and P are typable.
   The set of solutions of T is denoted by $\mathcal{S}ol(\mathsf{T})$.

Different theories and the corresponding pattern matching problems can be formally defined and solved, for example, as explained in [9]. By convention, if the solution of the equation $A \prec\!\!\prec_{\mathbb{T}} B$ is unique, it is denoted by $\theta_{(A \prec\!\!\prec_{\mathbb{T}} B)}$.

2.2 Operational Semantics of the General Rewriting Calculus, $\rho^{\mathbb{T}}$

By now we have settled all the background necessary to describe in Figure 2 the reduction rules of the general ρ-calculus, called $\rho^{\mathbb{T}}$, parameterized by the theory $\mathbb{T}$. When instantiating $\mathbb{T}$ with concrete theories (*e.g.* theories containing axioms for associativity, associativity-commutativity, *etc*, for a given symbol) different versions of the calculus are obtained. When not essential or clear from the context, we will omit the theory $\mathbb{T}$ in rules and congruences.

$$(P \twoheadrightarrow_{\Delta} A)\ B \rightarrow_{\rho} [P \ll_{\Delta} B]A$$

$$[P \ll_{\Delta} B]A \rightarrow_{\sigma} A\theta_1, \ldots, A\theta_n \qquad \text{with } \{\theta_1, \ldots, \theta_n\} = \mathcal{S}ol(P \prec\!\!\!\prec_{\mathbb{T}} B)$$

$$(A, B)\ C \rightarrow_{\delta} A\ C, B\ C$$

Fig. 2. Top-level Rules of the General Rewriting Calculus, $\rho^{\mathbb{T}}$

Let us quickly explain the top-level rules:

(ρ) this rule "fires" the application of an abstraction to a term, but does not immediately try to solve the associated matching equation.

(σ) this rule is applied if (and only if) the matching equation $P \prec\!\!\!\prec_{\mathbb{T}} B$ has at least one solution: in this case the matching solutions are computed and applied to the term A. If the matching is not unitary, a structure collecting all the different results is obtained when the rule is applied. If there is no solution, this rule does not apply and thus, the term represents a matching failure. As we will see, further reductions or instantiations are likely to modify B so that the equation has a solution and the rule can be fired.

(δ) this rule distributes structures on the left-hand side of the application. This gives the possibility, for example, to apply in parallel two distinct pattern abstractions A and B to a term C.

We denote by $\mapsto_{\rho\sigma\delta}$ the contextual closure of these rules. Its reflexive and transitive closure is denoted $\mapsto\!\!\!\!\rightarrow_{\rho\sigma\delta}$. The symmetric and transitive closure of $\mapsto\!\!\!\!\rightarrow_{\rho\sigma\delta}$ is denoted $=_{\rho\sigma\delta}$.

2.3 The Fixpoint Rewriting Calculus, $\rho^{\mathsf{stk}}_{\twoheadrightarrow}$

We present a version of the rewriting calculus that handles uniformly matching failures and eliminates them when not significant for the computation. We define the rules for handling this kind of terms and we show how these are integrated in the calculus.

We define first a superposition relation $\sqsubseteq : \mathcal{P} \times \mathcal{T}$ between (patterns and) terms whose aim is to characterize a broad class of matching equations that are *potentially* solvable. If $P \sqsubseteq A$ we say that "P *does potentially superpose with* A" and, by negation, if $P \not\sqsubseteq A$ then "P *surely does not superpose with* A" (*i.e.* independently of subsequent instantiations and reductions).

Definition 4 (Superposition).

1. *The relation $P \sqsubseteq A$ is defined as follows by cases on the structure of P:*

$$\begin{array}{lll} f \sqsubseteq f & \mathsf{stk} \sqsubseteq \mathsf{stk} & X \sqsubseteq A \ (\forall A) \end{array}$$

$$f\,\overline{A} \sqsubseteq B \quad \text{if } (B \equiv f\,\overline{B}) \wedge \overline{A} \sqsubseteq \overline{B}$$

$$P \sqsubseteq A \quad \text{if } A \equiv \begin{cases} X \vee (A_1, A_2) \vee (A_1\ A_2 \wedge A_1 \notin \mathcal{P}) \vee \\ ([Q \ll_{\Delta} A_1]A_2 \wedge Q \sqsubseteq A_1 \wedge P \sqsubseteq A_2) \ \ (\forall P) \end{cases}$$

2. *If $P \sqsubseteq A$ is not satisfied we write $P \not\sqsubseteq A$.*

Starting from this relation, we define a reduction that eliminates from a ρ-term all the definitive stuck subterms, *i.e.* all the delayed matching constraints whose matching problem is unsolvable independently of subsequent instantiations and reductions.

Definition 5 (Stuck Theory, $\mathbb{T}_{\mathsf{stk}}$).
The relation $\rightharpoonup_{\mathsf{stk}}$ is defined by the following rules:

$$[P \ll_{\Delta} A]B \rightharpoonup_{\mathsf{stk}} \mathsf{stk} \qquad \text{if } P \not\sqsubseteq A \tag{1}$$

$$\mathsf{stk}, A \rightharpoonup_{\mathsf{stk}} A \tag{2}$$

$$A, \mathsf{stk} \rightharpoonup_{\mathsf{stk}} A \tag{3}$$

$$\mathsf{stk}\ A \rightharpoonup_{\mathsf{stk}} \mathsf{stk} \tag{4}$$

We denote by $\mapsto_{\mathsf{stk}}$ the contextual closure of these rules. Its reflexive and transitive closure is denoted by $\mapsto\!\!\!\rightarrow_{\mathsf{stk}}$. The symmetric and transitive closure of $\mapsto\!\!\!\rightarrow_{\mathsf{stk}}$ is denoted by $\stackrel{\mathsf{stk}}{=}$. Let $\mathbb{T}_{\mathsf{stk}}$ be the theory associated to the congruence $\stackrel{\mathsf{stk}}{=}$. Matching equations in the theory $\mathbb{T}_{\mathsf{stk}}$ are denoted $P \prec\!\!\!\prec_{\mathsf{stk}} A$.

As mentioned previously, these rules are used to propagate or eliminate the definitive stuck terms:

– Structures can be seen as collections of results and thus we want to identify all the (matching) failures and eliminate them from these collections; this is done by the first rules $(1-3)$;
– On the other hand, a stk term can be seen as an empty set of results; the rule (4) corresponds then to the (δ) rule dealing with empty structures and thus, to a propagation of the failure.

Lemma 1 (Confluence and Termination of stk-reduction).
The reduction $\mapsto_{\mathsf{stk}}$ is confluent and terminating.

In general, matching modulo the $\mathbb{T}_{\mathsf{stk}}$ theory is obviously infinitary. When restricting to matching equations with an algebraic left-hand side we can still have an infinite number of solutions but a unique representative can be always characterized. Intuitively, in this latter case, the canonic solution of a matching equation is the solution obtained by a syntactic matching algorithm with all the terms reduced in $\mapsto_{\mathsf{stk}}$-normal form. For example, since the solution of the equation $f\ X \prec\!\!\!\prec_{\mathbb{T}_\emptyset} f\ (a, \mathsf{stk})$ is $\{(a, \mathsf{stk})/X\}$, the solution of $f\ X \ll_{\mathsf{stk}} f\ (a, \mathsf{stk})$ is $\{a/X\}$, representing a witness for all the solutions with the shape $\{(\mathsf{stk}, \ldots, a, \ldots, \mathsf{stk})/X\}$.

Thus, for the sake of simplicity and in order to keep closer to possible implementations, we define the underlying relation of the calculus:

Definition 6 (Semantics of $\rho_{\twoheadrightarrow}^{\mathsf{stk}}$).
The underlying relation of $\rho_{\twoheadrightarrow}^{\mathsf{stk}}$, denoted $\mapsto_{\rho\sigma\delta}^{\mathsf{stk}}$ is defined as the relation $\mapsto_{\mathsf{stk}} \cup \mapsto_{\rho\sigma\delta}$.

For $\mapsto_{\rho\sigma\delta}^{\mathsf{stk}}$, the following holds:

Theorem 1 (Church Rosser for $\rho_{\twoheadrightarrow}^{\mathsf{stk}}$).
The relation $\mapsto_{\rho\sigma\delta}^{\mathsf{stk}}$ is confluent.

3 The First-order Type System for $\rho_{\twoheadrightarrow}^{\mathsf{stk}}$

Figure 3 presents the typing rules of $\rho_{\twoheadrightarrow}^{\mathsf{stk}}$, which are directly inspired by the simply typed λ-calculus.

$$\frac{\alpha{:}\tau \in \Gamma \quad \tau \in \mathcal{T}\mathsf{y}}{\Gamma \vdash \alpha : \tau}\ (Start) \qquad \frac{\Gamma, \Delta \vdash P : \tau_1 \quad \Gamma, \Delta \vdash A : \tau_2}{\Gamma \vdash P \twoheadrightarrow_{\Delta} A : \tau_1 \twoheadrightarrow \tau_2}\ (Abs)$$

$$\frac{\tau \in \mathcal{T}\mathsf{y}}{\Gamma \vdash \mathsf{stk} : \tau}\ (Stuck) \qquad \frac{\Gamma \vdash A : \tau_1 \twoheadrightarrow \tau_2 \quad \Gamma \vdash B : \tau_1}{\Gamma \vdash A\ B : \tau_2}\ (Appl)$$

$$\frac{\Gamma \vdash A : \tau \quad \Gamma \vdash B : \tau}{\Gamma \vdash A, B : \tau}\ (Struct) \qquad \frac{\Gamma, \Delta \vdash P : \tau_1 \quad \Gamma \vdash B : \tau_1 \quad \Gamma, \Delta \vdash A : \tau_2}{\Gamma \vdash [P \ll_{\Delta} B]A : \tau_2}\ (Match)$$

Fig. 3. The Type System for $\rho_{\twoheadrightarrow}^{\mathsf{stk}}$

- $(Start)$**:** The context determines the type of variables and constants. It cannot contain two declarations for the same variable (or constant);
- (Abs)**:** As mentioned in Section 2.1, $\mathsf{Dom}(\Delta) = \mathsf{Fv}(P)$. For the left-hand side of the arrow-type, we use the type of the pattern P; notice that the (Abs) rule allows one to hide some type informations in a pattern containing applications, *e.g.* τ_2 disappear in the final type of $(f\ X)$ in the judgment $f{:}\tau_2 \twoheadrightarrow \tau_1, X{:}\tau_2 \vdash f\ X : \tau_1$.
- $(Appl)$**:** We directly exploit the information given in the type of the function, statically checking that the given argument has the expected type τ_1;
- $(Struct)$**:** This rule states that all the members of a structure have the same type. This is important when considering structures as a collection of results; if a function can return different results, we would at least expect them to have the same type;
- $(Stuck)$**:** Since stk can appear in any structure, it can have any type;
- $(Match)$**:** This rule states that the constraint $[P \ll_{\Delta} B]A$ gets the same type as $(P \twoheadrightarrow_{\Delta} A)\ B$. This is sound since $(P \twoheadrightarrow_{\Delta} A)\ B \to_{\rho} [P \ll_{\Delta} B]A$. Once again, $\mathsf{Dom}(\Delta) = \mathsf{Fv}(P)$.

Example 1 (Simple type derivation).
The $(Appl)$ rule is effective for the typing of algebraic terms too. Let $\Gamma \triangleq f{:}\iota \to \iota, a{:}\iota$.

$$\frac{\Gamma \vdash f : \iota \to \iota \quad \Gamma \vdash a : \iota}{\Gamma \vdash f\ a : \iota}\ (Appl)$$

and, let $\Gamma \triangleq f{:}\tau_1 \to \tau_2, g{:}\tau_1 \to \iota, a{:}\tau_1$.

$$\frac{\dfrac{\Gamma, X{:}\tau_1 \vdash f\ X : \tau_2 \quad \Gamma, X{:}\tau_1 \vdash g\ X : \iota}{\Gamma \vdash f\ X \to_{(X:\tau_1)} g\ X : \tau_2 \to \iota}\ (Abs) \quad \Gamma \vdash f\ a : \tau_2}{\Gamma \vdash (f\ X \to_{(X:\tau_1)} g\ X)\ (f\ a) : \iota}\ (Appl)$$

This type system has been designed as a typing discipline for a programming language: its aim is to ensure that the arguments of a function have the same types as the corresponding formal parameters. However, the notion of (well-typed) pattern used here is crucial since it guarantees that the instantiation of the variables of the pattern will be correct with respect to types (*i.e.* the substitutions obtained as result of the matching are well-typed) even if no type-checking is performed in the matching algorithm.

We state in what follows the main properties of the typed calculus.

Lemma 2 (Substitution Lemma).
If $\Gamma, \Delta \vdash A : \tau$, then for any substitution θ well-typed in Γ such that $\mathsf{Dom}(\theta) = \mathsf{Dom}(\Delta)$, we have $\Gamma \vdash A\theta : \tau$.

Theorem 2 (Subject Reduction for $\rho_{\to}^{\mathsf{stk}}$).
If $\Gamma \vdash A : \tau$ and $A \mapsto_{\rho\delta}^{\mathsf{stk}} B$, then $\Gamma \vdash B : \tau$.

Theorem 3 (Type Uniqueness for $\rho_{\to}^{\mathsf{stk}}$).
If $\Gamma \vdash A : \tau_1$ and $\Gamma \vdash A : \tau_2$, and $\mathsf{stk} \notin A$, then $\tau_1 \equiv \tau_2$.

Theorem 4 (Decidability of Typing for $\rho_{\to}^{\mathsf{stk}}$).
If $\mathsf{stk} \notin A$ and $\mathsf{Fv}(A) \subseteq \mathsf{Dom}(\Gamma)$, then the following problems are decidable:

1. *Type Reconstruction: for a given Γ, is there a type τ such that $\Gamma \vdash A : \tau$?*
2. *Type Checking: given a context Γ, and a type τ, is it true that $\Gamma \vdash A : \tau$?*

4 Examples and Applications in $\rho_{\to}^{\mathsf{stk}}$

It has already been shown that the ρ-calculus allows one to faithfully encode first order term rewriting [8] as well as some classical object-calculi [9]. In this section, we show that $\rho_{\to}^{\mathsf{stk}}$ is sufficiently expressive and flexible for a more concise encoding that does not break the type discipline for these two formalisms. In most of the section, some type decorations of variables and constants are omitted for the sake of readability.

4.1 Encoding Abadi and Cardelli's Object-Calculus

In this section we briefly describe an encoding in the typed $\rho_{\twoheadrightarrow}^{\mathsf{stk}}$ of the classical object-calculus ςObj [1]. By better exploiting the pattern matching facilities of the $\rho_{\twoheadrightarrow}^{\mathsf{stk}}$ we obtain a more concise representation than the one given in previous works for the untyped ρ-calculus. A method is encoded as $(m\ S) \twoheadrightarrow \mathcal{T}_m$[1], where the constant m is the name of the method, the variable S will play the role of the keyword this (containing a copy of the object itself) and $\mathcal{T}_m$ is a term encoding the body of the method. An object obj is then a structure filled with methods. The method $meth$ is then called by Kamin's self application [13] which says that $obj.meth \triangleq obj\ (meth\ obj)$:

$$
\begin{array}{rcl}
obj\ (meth\ obj) & \equiv & (\ldots, meth\ S \twoheadrightarrow \mathcal{T}_{meth}, \ldots)\ (meth\ obj) \\
& \mapsto_{\rho\sigma\delta} & \ldots,\ [meth\ S \ll meth\ obj]\mathcal{T}_{meth}\ , \ldots \\
& \mapsto\!\!\!\!\to_{\rho\sigma\delta}^{\mathsf{stk}} & \mathcal{T}_{meth}[obj/S]
\end{array}
$$

Observe that the other methods fail because the equation $(m\ S \prec\!\!\!\prec meth\ obj)$ has no solution for every $m \not\equiv meth$. The stk terms obtained for each of these method applications can be eliminated from the final result by successive $\mapsto_{\mathsf{stk}}$ steps. The variable S is indeed instantiated with obj in the body of the method, allowing all the usual operations on the keyword this.

As such, the previous example can be typed in the ρ-calculus as follows: lab is the constant type of labels, S has type $lab \twoheadrightarrow \tau$, and τ is the type of $\mathcal{T}_{meth}$. For the sake of simplicity, we suppose obj has just one method triggered by the constant $meth$, with type $(lab \twoheadrightarrow \tau) \twoheadrightarrow lab$.

$$
\frac{\Gamma \vdash meth\ S : lab \quad \Gamma \vdash \mathcal{T}_{meth} : \tau}{\Gamma \vdash meth\ S \twoheadrightarrow \mathcal{T}_{meth} : lab \twoheadrightarrow \tau}\ (Abs)
$$

Considering the meaning we want for S, it is sound that obj and S have the same type. Then $obj.meth \triangleq obj\ (meth\ obj)$ can be typed as follows (let $\Gamma \triangleq meth{:}(lab \twoheadrightarrow \tau) \twoheadrightarrow lab, \ldots$):

$$
\frac{\Gamma \vdash obj : lab \twoheadrightarrow \tau \quad \Gamma \vdash meth\ obj : lab}{\Gamma \vdash obj\ (meth\ obj) : \tau}
$$

We end this subsection with an object-oriented version of $\omega\omega$, showing that the divergence of object-oriented programs is somehow built in the self-application. Remember that $S.loop$ denotes the self application $S\ (loop\ S)$.

$$
\begin{array}{rcl}
\vdash\ \Omega & \triangleq & (loop\ S) \twoheadrightarrow S.loop : lab \twoheadrightarrow \tau \\
\Omega.loop & \equiv & (loop\ S \twoheadrightarrow S.loop)\ (loop\ \Omega) \\
& \mapsto\!\!\!\!\to_{\rho\sigma\delta} & \Omega.loop \\
& \mapsto\!\!\!\!\to_{\rho\sigma\delta} & \ldots
\end{array}
$$

[1] In [9], the original encoding was $m \twoheadrightarrow S \twoheadrightarrow \mathcal{T}_m$, needing two reduction steps where one is enough with our enhanced encoding.

4.2 Encoding Term Rewriting Systems (TRS)

The correspondence between the ρ-calculus and the TRS is not as straightforward as it may seem. Observe that a ρ-abstraction is consumed by a ρ-reduction, and therefore can operate only locally. For instance, the simple (one-rule) TRS consisting of $f(X) \to X$ reduces $f(f(f(a)))$ to a. In the ρ-calculus, we can have control over the application of this rule:

$$\begin{array}{rl} (f\ X \twoheadrightarrow X)\ (f\ (f\ (f\ a))) & \mapsto\!\!\!\!\rightarrow_{\rho\sigma\delta} f\ (f\ a) \\ (f\ Y \twoheadrightarrow Y)\ \big((f\ X \twoheadrightarrow X)\ (f\ (f\ (f\ a)))\big) & \mapsto\!\!\!\!\rightarrow_{\rho\sigma\delta} (f\ Y \twoheadrightarrow Y)\ (f\ (f\ a)) \mapsto\!\!\!\!\rightarrow_{\rho\sigma\delta} f\ a \end{array}$$

In general, encoding (first and higher order) rewriting systems in the untyped ρ-calculus requires a complex translation mechanism [8,4].

An (ad-hoc) Object-Oriented Encoding. We can define in the typed $\rho^{\mathsf{stk}}_{\twoheadrightarrow}$ a suitable self-duplicating term that allows us to simulate the *global* behavior of a TRS $\mathcal{R}$. Let us begin with the example of addition, using two constants $rec^{(lab \twoheadrightarrow \iota \twoheadrightarrow \iota) \twoheadrightarrow lab}$ and $add^{\iota \twoheadrightarrow \iota \twoheadrightarrow \iota}$.

Example 2 (Addition).

$$plus \triangleq rec\ S \twoheadrightarrow \begin{pmatrix} add\ 0\ Y \twoheadrightarrow Y, \\ add\ (suc\ X)\ Y \twoheadrightarrow suc\ \ (S.rec\ (add\ X\ Y)) \end{pmatrix}$$

Intuitively, the variable S acts like the meta-variable this in JAVA and thus, the recursive application of the different rules is realized explicitly by using this variable in the right-hand side of the corresponding rules.

This term computes indeed the addition over Peano integers, as illustrated in Fig. 4. The expressions "$\overline{m}$", and "$\overline{m+n}$", and "$\overline{m-n}$" are just aliases for the Peano representations of these numbers as sequences of $suc(\ldots(suc\ 0)\ldots))$. It is worth noticing that all the stuck results are dropped by $\mapsto_{\mathsf{stk}}$; the only interesting result is $[add\ 0\ Y \ll add\ 0\ \overline{m}]Y$. During the reduction, on the left of this term (or terms reducing to it) all the terms get stuck because we try to match 0 against $suc\ \overline{n}$; on the right too because we try to match $suc\ X$ against 0. Notice that if we erase from the term $plus$ all the "administrative" subterms which encode the recursive machinery, we get back a TRS computing addition:

$$\begin{array}{ll} add(0, Y) & \to Y \\ add(suc(X), Y) & \to suc\,(add(X, Y)) \end{array}$$

Observe that, in this rather ad-hoc encoding of $plus$, we have put $S.rec$ only before the symbol add in the right-hand side of the second rule because we know that it is the only position where further rewriting has to be done. In the next paragraph we describe a more general method for encoding a TRS.

An Automated Encoding Using the first Operator. In this paragraph, we show that any convergent and well-typed TRS $\mathcal{R}$ can be mechanically encoded (and typechecked) in $\rho^{\mathsf{stk}}_{\twoheadrightarrow}$. Recall that a TRS $\mathcal{R}$ is convergent if it is confluent and (strongly) terminating; $\mathcal{R}$ is well-typed if all the rewrite rules can be typechecked with the same type for both sides of the arrow. The encoding is done by wrapping $\mathcal{R}$ into a *typed object-based fixpoint*

$$
\begin{array}{ll}
 & plus.rec \;\; (add\, \overline{n}\, \overline{m}) \\
\mapsto\!\!\!\!\to_{\rho\delta} & (add\, 0\, Y \to Y) \;\; (add\, \overline{n}\, \overline{m}), \\
 & (add\, (suc\; X)\; Y \to suc\; (plus.rec \;\; (add\; X\; Y))) \;\; (add\, \overline{n}\, \overline{m}) \\
\mapsto\!\!\!\!\to_{\rho\delta} & [add\, 0\, Y \ll add\, \overline{n}\, \overline{m}]Y, \\
 & suc\, (plus.rec \;\; (add\, \overline{n-1}\, \overline{m})) \\
\mapsto\!\!\!\!\to_{\mathsf{stk}} & suc\, (plus.rec \;\; (add\, \overline{n-1}\, \overline{m})) \\
\mapsto\!\!\!\!\to_{\rho\delta} & suc\; \big((add\, 0\, Y \to Y) \;\; (add\, \overline{n-1}\, \overline{m}), \\
 & \quad (add\, (suc\; X)\; Y \to suc\; (plus.rec \;\; (add\; X\; Y))) \;\; (add\, \overline{n-1}\, \overline{m})\big) \\
 & \ldots \\
\mapsto\!\!\!\!\to_{\rho\delta} & suc\, ([add\, 0\, Y \ll add\, \overline{n{-}1}\, \overline{m}]Y, \\
 & \quad suc\, (\cdots suc\, ((add\, 0\, Y \to Y) \;\; (add\, 0\, \overline{m}), \\
 & \qquad\qquad (add\; (suc\; X)\; Y \to suc\; (plus.rec \;\; (add\; X\; Y))) \;\; (add\, 0\, \overline{m})))) \\
\mapsto\!\!\!\!\to_{\mathsf{stk}} & suc\, (suc\, (\cdots suc\, ((add\, 0\, Y \to Y) \;\; (add\, 0\, \overline{m}), \\
 & \qquad\qquad (add\; (suc\; X)\; Y \to suc\; (plus.rec \;\; (add\; X\; Y))) \;\; (add\, 0\, \overline{m})))) \\
\mapsto\!\!\!\!\to_{\rho\delta} & suc\, (suc\, (\cdots suc\, ([add\, 0\; Y \ll add\; 0\, \overline{m}]Y, \\
 & \qquad\qquad [add\; (suc\; X)\; Y \ll add\; 0\, \overline{m}](suc\; (plus.rec \;\; (add\; X\; Y)))))) \\
\mapsto\!\!\!\!\to_{\mathsf{stk}} & suc\, (suc\, (\cdots suc\, ([add\, 0\; Y \ll add\; 0\, \overline{m}]Y))) \\
\mapsto\!\!\!\!\to_{\rho\delta} & suc\, (suc\, (\cdots (suc\, \overline{m}))) \\
\triangleq & \overline{m{+}n}
\end{array}
$$

Fig. 4. A complete reduction for a ρ-term encoding addition

engine which is a ρ-term that encodes all the rules in $\mathcal{R}$ and applies the translated rules recursively until none of them is applicable. Definition 7 details how this ρ-term is built: the right-hand sides are modified so that the whole system can be re-applied to any of the symbols appearing in the term.

We first define the operator "first" that tries to apply successively n rules $A_1, A_2, \ldots A_n$ to a term B and returns the result of the first rule whose application succeeds (*i.e.* does not reduce to stk). Here we use the constant stk to detect the failure of a given rule and the identity $\mathsf{I} \triangleq Y \to Y$ to yield a successful result:

$$\mathsf{first}(A_1, A_2, \ldots, A_n) \;\triangleq\; X \to ((\mathsf{stk} \to A_n\; X, \mathsf{I}) \;\; (\ldots \;\; (\mathsf{stk} \to A_2\; X, \mathsf{I}) \;\, (A_1 X)))$$

One can check that when we reduce $\mathsf{first}(A, B)\; C$, if $A\; C$ reduces to stk then the final result is the reduct of the term $B\; C$, since the stk produced by $A\; C$ will be discarded by further $\mapsto\!\!\!\!\to_{\mathsf{stk}}$ reductions even if it is accepted by the identity. If $A\; C$ reduces to an algebraic term D different of stk, it passes through I and since the matching equation $\mathsf{stk} \not\ll D$ definitively fails (leading to a stk term), the final result is D. The same behavior is obtained for an arbitrary number of arguments for first. In particular, we will use the term $\mathsf{first}(A_1, \ldots, A_n, \mathsf{I})$, which tries successively the n rules $A_1, \ldots A_n$ on the argument B, and returns B unchanged if every $A_i\; B$ fails.

From a typed point of view, the behavior of first is easy to understand: every A_i can be applied to X, so each must have a type $\tau \to \tau_i$ where $X : \tau$. Moreover, for each

i, we apply $(\mathsf{stk} \rightharpoondown A_{i+1}, \mathsf{I})$ to $A_i X$. Here the identity I has type $\tau_i \rightharpoondown \tau_i$. Since all the members of a structure must have the same type, $A_{i+1}X$ has type τ_i too. By trivial induction, all the A_i have the same type, which is the type of $\mathsf{first}(A_1, \ldots, A_n)$. We can informally state this result by $\Gamma \vdash \mathsf{first} : (\tau \rightharpoondown \tau_0) \rightharpoondown \ldots \rightharpoondown (\tau \rightharpoondown \tau_0) \rightharpoondown \tau \rightharpoondown \tau_0$ where Γ is such that $\Gamma \vdash A_i : \tau \rightharpoondown \tau_0$ for each A_i.

In what follows, we denote by $s, t, \ldots$ algebraic terms (in the sense of the grammar $x \mid f(t, \ldots t)$) and a term rewriting system by $\mathcal{R} \triangleq \{t_i \to s_i\}^{i=1..n}$. We write $t \mapsto\!\!\!\twoheadrightarrow_{\mathcal{R}} t'$ when t can be rewritten to the term t' in normal form *w.r.t.* the TRS $\mathcal{R}$.

Definition 7 (Object-based fixpoint engine). *Let $\mathcal{R} = \{t_i \to s_i\}^{i=1..n}$ be an untyped TRS with terms built on a signature $\mathcal{F} = \{a_1, \ldots, a_m\}$; let S be a fresh variable w.r.t. $\mathcal{R}$. The encoding of $\mathcal{R}$ in $\rho^{\mathsf{stk}}_{\rightharpoondown}$ is done as follows:*

- *the terms t_i and s_i are transformed into ρ-terms using the translation $(\!|\, - \,|\!)$:*

$$\begin{aligned} (\!|\, x \,|\!) &= X \\ (\!|\, f(t_1, \ldots, t_n) \,|\!) &= f\ (\!|\, t_1 \,|\!)\ (\!|\, t_2 \,|\!) \cdots (\!|\, t_n \,|\!) \end{aligned}$$

- *the $\rho^{\mathsf{stk}}_{\rightharpoondown}$-term encoding $\mathcal{R}$ is denoted $(\!|\, \mathcal{R} \,|\!)$:*

$$(\!|\, \mathcal{R} \,|\!) \;\triangleq\; \begin{pmatrix} rec(S) \rightharpoondown \mathsf{first} \begin{pmatrix} (\!|\, t_1 \,|\!) \rightharpoondown S.rec\ (\!|\, s_1 \,|\!), \\ \cdots, \\ (\!|\, t_n \,|\!) \rightharpoondown S.rec\ (\!|\, s_n \,|\!), \\ a_1\ \overline{X} \rightharpoondown S.Rec\ (a_1\ \overline{S.rec\ X}), \\ \cdots, \\ a_m\ \overline{X} \rightharpoondown S.Rec\ (a_m\ \overline{S.rec\ X}) \end{pmatrix}, \\ Rec(S) \rightharpoondown \mathsf{first} \begin{pmatrix} (\!|\, t_1 \,|\!) \rightharpoondown S.rec\ (\!|\, s_1 \,|\!), \\ \cdots, \\ (\!|\, t_n \,|\!) \rightharpoondown S.rec\ (\!|\, s_n \,|\!), \\ \mathsf{I} \end{pmatrix} \end{pmatrix}$$

The result corresponding to the rewriting of an input term t w.r.t. to a TRS $\mathcal{R}$ is computed by the $\rho^{\mathsf{stk}}_{\rightharpoondown}$-term $(\!|\, \mathcal{R} \,|\!).rec\ (\!|\, t \,|\!)$.

This encoding enforces an *outermost* strategy: the ρ-term $(\!|\, \mathcal{R} \,|\!).rec$ first tries to apply a rule at top level, and if no one succeeds it uses the rules $(a_i\ \overline{S.rec\ X})$, $1 \leq i \leq m$ to propagate the TRS deeper in the term. The second method, called by $S.Rec$, no longer needs to propagate the TRS inside the term because it is used only when the subterms have been totally reduced, thus the only possibly reducible position is the head of the term. Some more subtle combinations of the different rules in the TRS could lead to various interesting strategies like, for example, innermost or call-by-need.

We prove that the encoding is faithful for TRS satisfying confluence and termination, which is often required in rewriting-based languages for (the part of the) rewriting systems that are not guided by a strategy.

Theorem 5 (Soundness and Completeness of the Rewriting Engine).

1. *For any TRS $\mathcal{R}$, and any algebraic input term t, if A is a $\rho^{\mathsf{stk}}_{\to}$-term in normal form w.r.t. $\mapsto\!\!\!\!\rightarrow^{\mathsf{stk}}_{\rho\sigma\delta}$ and without matching failures, then:*

$$(\!|\, \mathcal{R} \,|\!).rec \; (\!|\, t \,|\!) \mapsto\!\!\!\!\rightarrow^{\mathsf{stk}}_{\rho\sigma\delta} A \quad \Rightarrow \quad t \mapsto\!\!\!\!\rightarrow_{\mathcal{R}} t'$$

where $(\!|\, t' \,|\!) = A$.

2. *If the TRS $\mathcal{R}$ is* well-typed *and* convergent*, then for any algebraic terms t, t',*

$$t \mapsto\!\!\!\!\rightarrow_{\mathcal{R}} t' \;\Rightarrow\; (\!|\, \mathcal{R} \,|\!).rec \; (\!|\, t \,|\!) \mapsto\!\!\!\!\rightarrow^{\mathsf{stk}}_{\rho\sigma\delta} (\!|\, t' \,|\!).$$

Remark 1. These conditions are tight, in the sense that, for most of the non-confluent (or non-terminating) TRS, there is a term t and a reduction path $t \mapsto\!\!\!\!\rightarrow_{\mathcal{R}} t'$ which can not be mimicked by $(\!|\, \mathcal{R} \,|\!).rec \; (\!|\, t \,|\!)$. For the non-confluence this can be easily seen: our encoding enforces a particular strategy, so if two reduction paths are possible from a given term, then the engine has to choose one.

Thus, this encoding allows one to use the rewriting calculus in order to represent many well-typed TRS, and at the same time have a control over the correctness of the rules by means of a simple typechecking mechanism.

Example 3 (Computing the length of a list).

$$\begin{aligned} length \;\triangleq\; rec\ S \to \mathsf{first} & \begin{pmatrix} len\ nil & \to 0\ , \\ len\ (cons\ X\ L) & \to S.rec\ (suc\ (len\ L))\ , \\ suc\ X & \to S.Rec\ suc\ (S.rec\ X) \end{pmatrix}, \\ Rec\ S \to \mathsf{first} & \begin{pmatrix} len\ nil & \to 0\ \ , \\ len\ (cons\ X\ L) & \to S.rec\ (suc\ (len\ L)), \\ \mathsf{I} & \end{pmatrix} \end{aligned}$$

Type-checking all the Encodings. Each of the above encodings can be completed by a type-checking phase. The terms built in $\rho^{\mathsf{stk}}_{\to}$ can not be well-typed only if the initial rewriting system cannot be typed correctly.

$(plus)$ It is easy to check that the naive encoding of $plus$ can be type checked with $\Gamma \vdash plus{:}lab \to \iota \to \iota$, where $\Gamma \triangleq rec{:}(lab \to \iota \to \iota) \to lab, add{:}\iota \to \iota \to \iota, suc{:}\iota \to \iota, 0{:}\iota$.

$((\!|\, \mathcal{R} \,|\!))$ The object-based fixpoint engine has type $lab \to \tau \to \tau$ where τ is the type of the data manipulated by $\mathcal{R}$. Here rec and Rec both have type $(lab \to \tau \to \tau) \to lab$.
There must be a unique type τ for the data manipulated by the TRS: as we said before about first, all the A_i must have a common type $\tau \to \tau_0$. Since the identity I (applied if no rule in $\mathcal{R}$ can be used) has a type $\tau \to \tau$, all the rules in $\mathcal{R}$ must have the same type for their left and right-hand sides. This condition is not required in term rewriting systems, but it is generally imposed, for safety's sake, in most of the languages based on rewriting (*e.g.* Elan [12], Maude [16]).

$(length)$ Similarly to $plus$, the term $length$ type-checks with $\Gamma \vdash length : lab \to \iota \to \iota$, where $\Gamma \triangleq rec{:}(lab \to \iota \to \iota) \to lab, cons{:}\iota \to list \to list, len{:}list \to \iota, nil{:}list$. Notice that the type of the constant rec depends on the type of the data the TRS manipulates: we need in fact a whole class of rec constants (roughly one for each type) in order to write any fixpoint.

5 Conclusions : Related and Future Work

We have studied the expressive power of the simply typed $\rho^{\mathsf{stk}}_{\rightarrow}$. The defined type system was mainly adapted from the simply typed λ-calculus. The type system deals with all the particularities of the calculus (abstraction over arbitrary patterns, delayed matching constraints, structures of rules and results). The most interesting properties of typed calculi are valid: subject reduction, and type uniqueness, and decidability of typing.

Early versions of the (untyped) rewriting calculus have already been used to describe the implicit (leftmost-innermost) and user defined strategies of Elan [8] but some *ad hoc* operators were added to the basic calculus for this. The $\rho^{\mathsf{stk}}_{\rightarrow}$ presented here is a simpler formulation of the ρ-calculus essentially based on [11], where no new constructions have to be defined for expressing strategies for quite a large class of rewriting systems. The price to pay for this simplicity is that the encoded rewriting systems we handle are not the most general ones but still the most used ones in practice. The ability to typecheck term rewriting systems and strategies ensures a good trade-off between expressiveness and safety of programs.

Related Work. Some aspects of the relations between rewriting, λ-calculus and types have already been explored. V. van Oostrom has widely studied the confluence of a λ-calculus with patterns [17] but the presented language is untyped. D. Kesner, L. Puel and V. Tannen have proposed a typed pattern calculus [14] which has been designed as a computational interpretation of the Gentzen sequent proofs for the intuitionistic propositional logic. Our encoding of TRS shares some similarities with the one presented by S. Buyn *et al.* [6] that describes an untyped encoding of every strongly separable orthogonal TRS into λ-calculus.

The type system of $\rho^{\mathsf{stk}}_{\rightarrow}$ presented in this paper is quite different from the ones recently presented in the literature [10,3] since $\rho^{\mathsf{stk}}_{\rightarrow}$ does not use dependent types and thus patterns do not occur in types. In particular, the results about uniqueness, decidability and non-normalization *cannot* be transposed straightforwardly to those (logic-oriented) type systems. Again, the main objective of $\rho^{\mathsf{stk}}_{\rightarrow}$ is to set the typed theoretical framework for a programming language featuring sophisticated and user customizable pattern matching facilities.

Some similar ways of producing non-normalization appear in various formalisms. N. P. Mendler [15] has shown that, when introducing recursive definitions in the typed λ-calculus, strong normalization is no longer enforced by typing if the type constructors do not satisfy a *"positiveness condition"*. This kind of condition is still present in the Calculus of Inductive Constructions which is the basis of the Coq proof assistant. The issue appears in programming languages too: for instance, in ML, one can define any recursive function without using the keyword let rec.

Therefore, the type system of $\rho^{\mathsf{stk}}_{\rightarrow}$ is suitable for static analysis, *i.e.* it ensures that functions get arguments of the expected type. However, as a wanted feature it does not enforce termination of the typed terms. We have shown the encoding of some interesting terms leading to infinite reductions by the use of the pattern matching features of the calculus. The consequence is that our typing discipline fits for a programming language since we are interested in type consistency and in recursive (potentially non-terminating) programs. Conversely, it is not adapted for defining a *Logical Framework*, since normalization is strongly linked to consistency, and it definitively differs from previous proposals of the authors [10,3].

Acknowledgments. The authors are sincerely grateful to Claude Kirchner for many fruitful discussions and invaluable comments about this work.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Verlag, 1996.
2. H. Barendregt. *Lambda Calculus: its Syntax and Semantics*. North Holland, 1984.
3. G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure Patterns Type Systems. In *Proc. of POPL*, pages 250–261. The ACM press, 2003.
4. C. Bertolissi, H. Cirstea, and C. Kirchner. Translating Combinatory Reduction Systems into the Rewriting Calculus. In *Proc. of RULE*. ENTCS, 2003.
5. P. Borovansky, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 2(285):155–185, 2002.
6. S. Byun, J. Kennaway, V. van Oostrom, and F. de Vries. Separability and Translatability of Sequential Term Rewrite Systems into the Lambda Calculus. Technical Report tr-2001-16, University of Leicester, 2001.
7. H. Cirstea and C. Kirchner. The Simply Typed Rewriting Calculus. In *Proc. of WRLA*. ENTCS, 2000.
8. 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.
9. H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In *Proc. of RTA*, volume 2051 of *LNCS*, pages 77–92. Springer-Verlag, 2001.
10. H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In *Proc. of FOSSACS*, volume 2030 of *LNCS*, pages 166–180, 2001.
11. H. Cirstea, C. Kirchner, and L. Liquori. Rewriting Calculus with(out) Types. In *Proc. of WRLA*, volume 71 of *ENTCS*, 2002.
12. Équipe Protheo. The Elan Home Page, 2003. `http://elan.loria.fr`.
13. S. N. Kamin. Inheritance in Smalltalk-80: A Denotational Definition. In *Proc. of POPL*, pages 80–87. The ACM press, 1988.
14. D. Kesner, L. Puel, and V. Tannen. A Typed Pattern Calculus. *Information and Computation*, 124(1):32–61, 1996.
15. N. P. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, USA, 1987.
16. The Maude Team. The Maude Home Page, 2003. `http://maude.cs.uiuc.edu/`.
17. V. van Oostrom. Lambda Calculus with Patterns. Technical Report IR-228, Faculteit der Wiskunde en Informatica, Vrije Universiteit Amsterdam, 1990.