The simply typed rewriting calculus

Horatiu Cirstea

LORIA and UHP
Nancy, France
Email: Horatiu.Cirstea@loria.fr

Claude Kirchner

LORIA and INRIA
Nancy, France
Email: Claude.Kirchner@loria.fr

Abstract

The *rewriting calculus* is a rule construction and application framework. As such it embeds in a uniform way term rewriting and lambda-calculus. Since rule application is an explicit object of the calculus, it allows us also to handle the set of results explicitly.

We present a simply typed version of the rewriting calculus. With a good choice of the type system, we show that the calculus is type preserving and terminating, i.e. verifies the subject reduction and strong normalization properties.

1 Introduction

The rewriting calculus [CK99a,CK99b] is a general framework handling explicitly the three notions of rule formation, rule application and rule application result. The rule formation constructor is denoted $\rightarrow$ and given two terms like $x+s(y)$ and $s(x+y)$ it allows us to built the rewrite rule $x+s(y) \rightarrow s(x+y)$. Applying the previous rewrite rule to the top position of the sum $3+2$ is performed using the application operator denoted $[\,](\,)$ (where 3 is an abbreviation for $s(s(s(0)))$). The result of the application $[x+s(y) \rightarrow s(x+y)](3+2)$ is the set $\{s(3+1)\}$. Why we handle sets of results could be understood simply by the fact that rewriting may either fail or give several results. For example when applying the previous rewrite rule on 0, the set of results is just the empty set: $[x+s(y) \rightarrow s(x+y)](0) = \emptyset$. If we assume furthermore the addition to be commutative, a rule application may lead to several terms like in $[x+s(y) \rightarrow s(x+y)](3+2) = \{s(3+1), s(2+2)\}$. All together, this so called ρ-calculus, has his objects (the ρ-terms) built using rule formation, rule application and sets, with no a priori restriction on the term formation.

It is of course important to provide a typed version of the calculus in order, in particular, to insure termination of the evaluation of well-typed ρ-terms. When dealing with the combination of term rewriting and λ-calculus, the first such result was obtained in [GBT89] and [Oka89]. One of the originalities of the ρ-calculus with respect to the previous approaches is that we consider only one calculus by opposition to the situation where the two frameworks of λ-calculus and rewriting are combined. We are here in a different situation where rewriting (and not the normalization process) is treated at the same level as λ-abstraction.

We have introduced and studied in [CK99b] the untyped rewriting calculus and we present here a simply typed version of it. We briefly present in the next section the ρ-calculus and some of its properties. In Section 3 we define the typed ρ-calculus. We then show the subject reduction property in Section 4 as well as strong normalization of the typed calculus in Section 5. The main ideas of the approach are the same as for λ-calculus but the two main difficulties consist in the correct handling of the matching process and the non-determinism of the calculus. This is allowed by the right definition of the typing system and by the careful handling of term and context sets in the proofs.

We use the same notations as in [CK99b] to which the reader is referred for details and examples about the untyped ρ-calculus and its application on the modeling of rewrite rules and strategies.

2 An informal presentation of the ρ-calculus

The ρ-calculus is defined by its five components: the *syntax* that makes precise the formation of the objects manipulated by the calculus, the description of the *substitution application* on terms, the *matching algorithm* used to bind variables to their actual values, the *evaluation rules* describing the way the calculus operates and the *strategy* guiding the application of the evaluation rules.

The core of the object formation in the ρ-calculus relies on a first-order signature together with rewrite rules formation, rule application and sets of results. The substitution application is often described at the meta-level of the calculus, except for explicit substitution frameworks. For the description of the ρ-calculus that we give here, we use (higher-order) substitutions and not grafting [DHK00], i.e. the application takes care of variable bindings and therefore uses α-conversion. In the general case, we consider a higher-order matching, but in practical cases it will be higher-order-pattern matching, or equational matching, or simply syntactic matching. The evaluation rules mainly describe the way a ρ-term is applied on another ρ-term and the the way sets are handled. Depending on the strategy employed for guiding the evaluation rules, we obtain different versions and therefore different properties for the calculus.

The rewriting calculus appears to be a natural way to express both term rewriting as well as λ-terms and their evaluation. A beta-redex $(\lambda x.t\ u)$ is nothing more than the ρ-term $[x \rightarrow t](u)$ (i.e. the application of the rewrite rule $x \rightarrow t$ on the term u) and while the above beta-redex reduces to $\{x/u\}t$, the corresponding ρ-redex reduces to $\{\{x/u\}t\}$ where $\{x/u\}t$ represents in both cases a higher order substitution application involving possible α-conversions. The λ-calculus with patterns presented in [PJ87] can be given a direct representation in the ρ-calculus. Let us consider, for example, the λ-term $\lambda(PAIR\ x\ y).x$ that selects the first element of a pair and the application $\lambda(PAIR\ x\ y).x\ (PAIR\ a\ b)$ that evaluates to a. The representation in the ρ-calculus of the first λ-term is $Pair(x, y) \rightarrow x$, where $Pair$ is the function symbol that corresponds to the symbol $PAIR$, and the application $[Pair(x, y) \rightarrow x](Pair(a, b))$ ρ-evaluates to $\{\{x/a, y/b\}x\}$, that is to $\{a\}$.

When building ρ-abstractions, i.e. rewrite rules, there is a priori no restriction. The application, in term rewriting, of a rewrite rule $f(y) \rightarrow g(a, y)$ to a term $f(b)$ is represented by the ρ-term $[f(y) \rightarrow g(a, y)](f(b))$ that evaluates to $\{g(a, b)\}$. But we may also rewrite an object into a rewrite rule like in the application $[x \rightarrow (f(y) \rightarrow g(x, y))](a)$ that evaluates to the singleton $\{f(y) \rightarrow g(a, y)\}$. In this case the variable x is free in the rewrite rule $f(y) \rightarrow g(x, y)$ but is bound in the rule $x \rightarrow (f(y) \rightarrow g(x, y))$. More generally, the object formation in ρ-calculus is unconstrained. Thus, the application of the rule $b \rightarrow c$ after the rule $a \rightarrow b$ on the term a is written $[b \rightarrow c]([a \rightarrow b](a))$ and as expected, the evaluation mechanism will produce first $[b \rightarrow c](\{b\})$ and then $\{c\}$. It also allows us to make use in an explicit and direct way of non-terminating or non-confluent (equational) rewrite systems. For example the application of the rule $a \rightarrow a$ on the term a (i.e. $[a \rightarrow a](a)$) terminates since it is applied only once and does not create a new redex. The set terms allow us to represent the non-deterministic application of a set of rules. For example, the ρ-term $[\{a \rightarrow b, a \rightarrow c\}](a)$ represents the non-deterministic application of one of the two rules and the result of the evaluation is the set $\{b, c\}$ representing the non-deterministic choice between the two results.

A ρ-calculus term contains all the (rewrite rule) information needed for its evaluation. This is also the case for λ-calculus but it is quite different from the usual way term rewrite *relations* are defined.

The rewrite relation generated by a rewrite system $\mathcal{R} = \{l_1 \rightarrow r_1, \ldots, l_n \rightarrow r_n\}$ is defined as the smallest transitive relation stable by context and substitution and containing $(l_1, r_1), \ldots, (l_n, r_n)$. For example, if we consider the rewrite system $\mathcal{R} = \{a \rightarrow f(a)\}$, then the relation contains $(a, f(a))$, $(a, f(f(a)))$, $(f(a), f(f(a))), \ldots$ and one says that the derivation $a \rightarrow f(a) \rightarrow f(f(a)) \rightarrow \ldots$ is generated by $\mathcal{R}$.

In ρ-calculus the situation is different since ρ-evaluation will reduce a given ρ-term in which all the rewriting information is explicit. It is customary to say that the rewrite system $a \rightarrow a$ is not terminating because it generates the derivation $a \rightarrow a \rightarrow a \rightarrow \ldots$. In ρ-calculus the same infinite derivation should be explicitly built (for example using an iterator) and all the evaluation information should be present in the starting term like in the ρ-term $[a \to a]([a \to a]([a \to a](a)))$ that could be used as a ρ-representation whose evaluation corresponds to the three steps derivation $a \to a \to a \to a$.

There is thus a big difference between the way one can define rewrite derivations generated by a rewrite system and their representation in ρ-calculus: in the first case the derivation construction is implicit and left at the meta-level, in the later case, all rewrite steps should be explicitly built.

To summarize, in ρ-calculus abstraction is handled via the arrow binary operator, matching is used as the parameter passing mechanism, substitution takes care of variable bindings and results sets are handled explicitly. Thus, the ρ-calculus integrates the matching power of rewriting and the higher-order functionality of λ-calculus and handles explicitly the non-determinism in the sense of sets of results.

The ρ-calculus is not confluent in the general case and the use of sets for representing the reduction results is the main cause of non-confluence. The confluence can be recovered if the evaluation rules of ρ-calculus are guided by an appropriate strategy [Cir00] handling properly the propagation of the failure (i.e. $\emptyset$) and the sets with more than one element.

The ρ-calculus is both conceptually simple as well as very expressive. This allowed us to represent the terms and reductions from λ-calculus and conditional rewriting. Using appropriate ρ-definitions for term traversal operators and a fixed point operator we are able to apply repeatedly a (set of) rewrite rule(s) and consequently to define a ρ-term representing the normalization according to a set of rewrite rules. Starting from this representation we showed how the ρ-calculus [CK99a,Cir00] can be used to give a semantics to ELAN [KKV93,BKK^{+}98] [1] rules and strategies. This could be applied to many other frameworks, including rewrite based languages like ASF+SDF [Deu96], ML [Mil84], Maude [CELM96], CafeOBJ [FN97] or Stratego [Vis99] but also production systems and non-deterministic transition systems.

In the context of rewriting logic [Mes92], proof terms are a subset of ρ-terms but there exist ρ-terms that do not correspond to any proof in rewriting logic. This extends the classical representation of proof terms by λ-terms used in logical frameworks by a representation using ρ-terms, therefore permitting us to use matching and non-determinism in the description of tactics and tacticals.

3 The typed ρ-calculus

The classical notations and definitions from this paper are inspired from those used for the typed λ-calculus used in [Hin97] and [HS86].

[1] A detailed description of ELAN can be found at `http://elan.loria.fr/`.

3.1 Syntax of the typed ρ-calculus

Let us consider a set $\mathcal{K}$ of *atomic types* $K_1, K_2, \ldots$ and the set of *types* inductively defined by:

- each *atomic type* is a *type*,
- if A and B are *types*, then $A \rightarrowtail B$ is a *type*,

The arrow of type definitions associates to right and thus a type of the form $A_1 \rightarrowtail A_2 \rightarrowtail \ldots \rightarrowtail A_n$ is an abbreviation for $A_1 \rightarrowtail (A_2 \rightarrowtail (\ldots \rightarrowtail A_n) \ldots)$.

The *atomic types* are intended to denote a particular set like, for example, the naturals or the booleans. The *compound types* of the form $A \rightarrowtail B$ are intended to denote the set of ρ-terms that can be applied to ρ-terms of type A giving as result ρ-terms of type B. Indeed, we also call *strategy* a ρ-term of *compound type.*

Definition 3.1 For a type A, a *typed variable* is denoted $x : A$ and we say that the variable x has the type A. A *context* is a *set* of typed variables. The assignments $x : A$ from a *context* are also called *variable type definitions (assignments).*

Definition 3.2 We consider $\mathcal{X}$ a set of variables and $\mathcal{F} = \bigcup_m \mathcal{F}_m$ a set of ranked first order function symbols possibly annotated with their rank. We denote by $\mathcal{T}(\mathcal{F}, \mathcal{X})$ the set of first order terms built on $\mathcal{F}$ using the variables in $\mathcal{X}$. If we denote by K any *atomic type*, then the syntax of the simply typed $\rho_\emptyset$-calculus is defined recursively by the following context-free grammar:

Types	T	$::=$	$K \mid T \rightarrowtail T$
Contexts	E	$::=$	$x : T \mid E \cdot \ldots \cdot E$
Terms	t	$::=$	$x \mid f(t, \ldots, t) \mid \{t, \ldots, t\} \mid u_{[E]} \rightarrow t \mid [t](t)$

where $x \in \mathcal{X}$, $u \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $f \in \mathcal{F}$.

The set containing no elements (i.e. $\{\}$) is also denoted by $\emptyset$. The contexts $E_1 \cdot \ldots \cdot E_n$ with $n = 0$ are represented by $\emptyset$. A context E restricted to the set of variables of a term t is denoted by $E|_t$.

The local context E of a rewrite rule $u_{[E]} \rightarrow t$ can contain any variables but we will see that the local context of a well-typed rewrite rule should contain exactly the typed variables concerned by the abstraction, i.e. the free variables of u.

For disambiguation purposes each symbol function could be annotated with its rank but when it is clear from the context this annotation is omitted.

If $A_1, \ldots, A_n, A$ are types we denote by $\mathcal{F}_{A_1 \times \ldots \times A_n \rightarrowtail A}$ the set of function symbols taking n arguments of type A_i and giving as result a term of type A. When a function symbol f belongs to such a set we denote it by $f \in \mathcal{F}_{A_1 \times \ldots \times A_n \rightarrowtail A}$. We overload the function symbols and consider for any symbol $f \in \mathcal{F}$ that if $f \in \mathcal{F}_{A_1 \times \ldots \times A_n \rightarrowtail A}$ and $f \in \mathcal{F}_{B_1 \times \ldots \times B_n \rightarrowtail B}$ then we also have the inclusions $f \in \mathcal{F}_{(A_1 \rightarrowtail B_1) \times \ldots \times (A_n \rightarrowtail B_n) \rightarrowtail (A \rightarrowtail B)}$ as well as

$f \in \mathcal{F}_{(B_1 \rightarrowtail A_1)\times\ldots\times(B_n \rightarrowtail A_n)\rightarrowtail(B \rightarrowtail A)}$. Additionally, if we have one of the ranks $f \in \mathcal{F}_{(A_1 \rightarrowtail B_1)\times\ldots\times(A_n \rightarrowtail B_n)\rightarrowtail(A \rightarrowtail B)}$ or $f \in \mathcal{F}_{(B_1 \rightarrowtail A_1)\times\ldots\times(B_n \rightarrowtail A_n)\rightarrowtail(B \rightarrowtail A)}$ then $f \in \mathcal{F}_{A_1\times\ldots\times A_n \rightarrowtail A}$ and $f \in \mathcal{F}_{B_1\times\ldots\times B_n \rightarrowtail B}$. The reason for this overloading will become apparent in the evaluation rule *Congruence* presented in Figure 3 where the function symbols with the same name should have different ranks in order to obtain well-typed terms and type preservation.

Definition 3.3 The set of *free variables* of a ρ-term t is denoted by $FV(t)$ and is defined by:

(i) if $t = x$ then $FV(t) = \{x\}$,

(ii) if $t = \{u_1, \ldots, u_n\}$ then $FV(t) = \bigcup_{i=1,\ldots,n} FV(u_i)$,

(iii) if $t = f(u_1, \ldots, u_n)$ then $FV(t) = \bigcup_{i=1,\ldots,n} FV(u_i)$,

(iv) if $t = [u](v)$ then $FV(t) = FV(u) \cup FV(v)$,

(v) if $t = u_{[\![E]\!]} \to v$ then $FV(t) = FV(v) \setminus FV(u)$.

The *bound variables* of a rewrite rule are the free variables of its left-hand side.

Definition 3.4 We say that a context is *consistent* if it does not contain two different variable type definitions for the same variable.

3.2 The typing rules of the ρ-calculus

The typing rules of the calculus are presented in Figure 1 where all the contexts are supposed to be consistent.

Definition 3.5 Given a formula $E \vdash t : A$ deduced using the typing rules of the ρ-calculus and a context E' such that $E \subseteq E'$ we say that the term t is *typeable* (or *well-typed*) and has the type A in the context E' and we denote it by $E' \vdash t : A$. We say that a term t is *typeable* in a context E' if there exists a type A such that t has the type A in the context E'. A term t is *typeable* if there exists a context in which t is typeable.

From the rule *Op* one can notice that the type of a term with a first order head symbol depends on the rank of the symbol and on the types of its arguments. We should point out that the typing rule *Op* works for constants (ground first order terms without arguments) as well and thus, if $a \in \mathcal{F}_A$ then $\emptyset \vdash a : A$.

The rule *Set* says that a set of terms is well-typed if all its elements have the same type. The empty set can be given any type.

When typing a rewrite rule $l \to r$ using the typing rule *Rule*, we consider the fact that the free variables from the right-hand side of the rule are bound by the free variables from the left-hand side. Due to this strong relationship between the variables with the same name from the two sides of the rewrite rule, the same context $(E|_l)$ should be used in order to give types to these variables. Furthermore, since in the $\rho_\emptyset$-calculus the left-hand side of a rewrite

Var	$E \vdash x : A$	$if\ x : A \subseteq E$
Rule	$\dfrac{E \vdash l : A \quad E\|_l \cdot F \vdash r : B}{F \vdash (l_{[\![E\|_l]\!]} \to r) : A \rightarrowtail B}$	
App	$\dfrac{E \vdash u : A \rightarrowtail B \quad E \vdash v : A}{E \vdash [u](v) : B}$	
Op	$\dfrac{E \vdash t_1 : A_1 \ldots E \vdash t_n : A_n}{E \vdash f(t_1, \ldots, t_n) : A}$	$if\ f \in \mathcal{F}_{A_1 \times \ldots \times A_n \rightarrowtail A}$
Set	$\dfrac{E \vdash t_1 : A \ldots E \vdash t_n : A}{E \vdash \{t_1, \ldots, t_n\} : A}$	
Empty	$E \vdash \emptyset : A$	*for any type A*

Fig. 1. Typing rules for the ρ-calculus

rule $l \to t$ is always a first-order term, the variables of the term l are exactly the free variables of l and therefore, the bound variables of the rewrite rule. Thus, the context allowing us to type a rewrite rule $l \to t$ does not have to include these typed variables (i.e $E|_l$) but should make precise the types of the free variables of r that are not free in l (i.e F).

3.3 Discussion on the typing of rewrite rules

One should notice that, according to Definition 3.5, the set of free variables of the left-hand side of a rewrite rule $l \to r$ is not necessarily the same as the set of typed variables from the context allowing us to type the term l. This latter context can contain variables that do not belong to l but free in r and thus, variables that should be in the global context used for typing the rewrite rule. In the typing rule *Rule*, the restriction of the context E to the set of variables of l avoids the elimination of variables that are not free in l and thus, not bound in the rule $l \to r$, from the context used for typing this rewrite rule.

Suppose, for example, that in the typing rule *Rule* the context $E|_l$ is replaced by E:

$$Rule' \ \frac{E \vdash l : A \quad E \cdot F \vdash r : B}{F \vdash (l_{[\![E]\!]} \to r) : A \rightarrowtail B}$$

Using this new rule we can infer $\emptyset \vdash x_{[\![x:A \cdot y:A]\!]} \to y : A \rightarrowtail A$ and if $a \in \mathcal{F}_A$ we obtain $\emptyset \vdash [x_{[\![x:A \cdot y:A]\!]} \to y](a) : A$. We will see that this latter application reduces in the ρ-calculus to $\{y\}$ and it is obvious that we cannot infer $\emptyset \vdash \{y\} : A$ and thus, the types are not preserved by reduction.

We can try to simplify the typing rule *Rule* and do not eliminate the typed variables of the left-hand side from the context of the rewrite rule and use the typing rule:

$$\mathit{Rule}'' \ \frac{E \vdash l : A \quad F \vdash r : B}{F \vdash (l_{\llbracket E \rrbracket} \to r) : A \rightarrowtail B}$$

In this case we should impose an explicit consistence condition on the context $E \cdot F$. If this condition is not satisfied we can obtain bound variables in the right-hand side of a rewrite rule that do not have the same type as the corresponding ones from the left-hand side of the respective rewrite rule.

For example, if we consider $x : A \vdash x : A$ and $x : B \vdash x : B$ then, by using the typing rule *Rule*$''$, we can infer $x : B \vdash x_{\llbracket x:A \rrbracket} \to x : A \to B$ and thus, we obtain the term $x_{\llbracket x:A \rrbracket} \to x$ that we obviously do not want to have the type $A \to B$ but $A \to A$.

On the other hand, the consistence of the context $E \cdot F$ is a too restrictive condition that would not allow us to type all the ρ-terms well-typed according to the typing rules in Figure 1.

Let us consider a function symbol $f \in \mathcal{F}_{B \rightarrowtail A}$. We can easily obtain $x : A \vdash x_{\llbracket x:A \rrbracket} \to x : A \to A$ but, although we have $x : B \vdash f(x) : A$, the term $[x_{\llbracket x:A \rrbracket} \to x](f(x))$ cannot be typed in the context $x : B$ due to the inconsistence of the context $x : A \cdot x : B$. Though, the term $[y_{\llbracket y:A \rrbracket} \to y](f(x))$ is well-typed in the context $x : B$ and we can say that all terms typed using the rules in Figure 1 can be typed modulo α-conversion using an approach based on the typing rule *Rule*$''$.

Additionally, since according to the rule *Rule*$''$, the context of a rewrite rule contains the typed variables of its left-hand side, we do not have to store these variables and the typing rule becomes:

$$\mathit{Rule}''' \ \frac{F \vdash l : A \quad F \vdash r : B}{F \vdash (l \to r) : A \rightarrowtail B}$$

Although, the initial typing rule *Rule* is slightly more difficult to handle we have prefered this approach that do not impose any restrictions on the terms that can be well-typed.

3.4 Typed substitutions

At this moment we should define typed substitutions and the way they apply to a typed term.

The *typed substitutions* are defined in the same way as the untyped substitutions (not to be confused with grafting or first order substitutions) but the types of the variables from the domain of the substitution are given explicitly. If $A_1, \ldots, A_n$ are types, a typed substitution has the form $\sigma = \{x_1 : A_1/t_1, \ldots, x_n : A_n/t_n\}$ with $x_i \in \mathcal{X}$ and t_i typed ρ-terms. The domain of the substitution σ is defined as usually: $Dom(\sigma) = \{x_1 : A_1, \ldots, x_n : A_n\}$.

Definition 3.6 A typed *substitution* $\sigma = \{x_1 : A_1/t_1, \ldots, x_n : A_n/t_n\}$ is *well-typed* in a context E and we denote it by $E \vdash \sigma$ if for all typed variables $x_i : A_i \in \mathit{Dom}(\sigma)$ we have $E \vdash t_i : A_i$.

The application of a well-typed substitution on a typed term is defined similarly as for the untyped case up to the condition that the domain of the substitution and the context of the term should be consistent:

Lemma 3.7 Given a term t, a typed substitution σ and the context E such that $E \vdash \sigma$ and $\mathit{Dom}(\sigma) \cdot E \vdash t : B$, with $\mathit{Dom}(\sigma) \cdot E$ consistent, then $E \vdash \sigma t : B$.

3.5 Typed matching

Computing the matching substitutions from a ρ-term t to a ρ-term t' is an important parameter of the calculus. For the purpose of this paper we define matching problems:

Definition 3.8 For a given theory $\mathcal{T}$ over ρ-terms, a *$\mathcal{T}$-match-equation* is a formula of the form $E \vdash t \ll^?_{\mathcal{T}} E' \vdash t'$ where t and t' are well-typed ρ-terms in the respective contexts E and E'. A well-typed substitution σ in a context E' is a solution of the $\mathcal{T}$-match-equation $E \vdash t \ll^?_{\mathcal{T}} E' \vdash t'$ if $\mathcal{T} \models \sigma(t) = t'$. A *$\mathcal{T}$-matching system* is a conjunction of $\mathcal{T}$-match-equations. A substitution is a solution of a $\mathcal{T}$-matching system P if it is a solution of all the $\mathcal{T}$-match-equations in P. We denote by $\mathbf{F}$ a $\mathcal{T}$-matching system without solution.
We define the function S*olution* on a $\mathcal{T}$-matching system P as returning the set of all $\mathcal{T}$-matches of P.

According to the previous definition, if the substitution σ is a solution of the match-equation $E \vdash t \ll^?_{\emptyset} E' \vdash t'$ and $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ then $\mathit{Dom}(\sigma) \subseteq E$.

For example when $\mathcal{T}$ is empty, the syntactic matching substitution from t to t', when it exists, is unique and can be computed, in the untyped case, by a simple recursive algorithm given for example by G. Huet [Hue76]. It can also be computed by the following set of rules *SyntacticMatching* where the symbol $\wedge$ is assumed to be associative and commutative.

We denote $\mathcal{M}atch(t, t', E, E')$ the substitution obtained by matching the term t that is well-typed in the context E against the term t' that is well-typed in the context E'.

Proposition 3.9 The normal form with regards to the matching rules in the set *SyntacticMatching* of any matching problem $t \ll^?_{\emptyset} t'$ exists and is unique. After removing from the normal form any duplicated match-equation, if the resulting system is:

(i) $\mathbf{F}$, then there is no match from t to t' and we have $\mathcal{M}atch(t, t', E, E') = \emptyset$,

(ii) of the form $\bigwedge_{i \in I} x_i \ll^?_{\emptyset} t'_i$ with $I \neq \emptyset$, then the substitution $\sigma = \{x_i/t'_i\}_{i \in I}$ is the unique match from t to t'. If $E \vdash x_i : A_i$ and $E' \vdash t'_i : A_i$ then

$$
\begin{array}{lll}
\mathit{Decomposition} & (f(t_1,\ldots,t_n) \ll^?_\emptyset f(t'_1,\ldots,t'_n)) \wedge P \mapsto\!\!\!\!\rightarrow & \\
 & \bigwedge_{i=1\ldots n} t_i \ll^?_\emptyset t'_i \wedge P & \\
 & & \\
\mathit{SymbolClash} & (f(t_1,\ldots,t_n) \ll^?_\emptyset g(t'_1,\ldots,t'_m)) \wedge P \mapsto\!\!\!\!\rightarrow & \mathbf{F} \\
 & & \text{if } f \neq g \\
\mathit{MergingClash} & (x \ll^?_\emptyset t) \wedge (x \ll^?_\emptyset t') \wedge P \mapsto\!\!\!\!\rightarrow & \mathbf{F} \\
 & & \text{if } t \neq t' \\
\mathit{VarClash} & (f(t_1,\ldots,t_n) \ll^?_\emptyset x) \wedge P \mapsto\!\!\!\!\rightarrow & \mathbf{F} \\
 & & \text{if } x \in \mathcal{X}
\end{array}
$$

Fig. 2. *SyntacticMatching* - Rules for syntactic matching

$\mathcal{M}atch(t,t',E,E') = \{\{x_i : A_i/t'_i\}_{i\in I}\}$. If for some $i \in I$ we have a variable $E \vdash x_i : A_i$ and $E' \vdash t'_i : B_i$ with $A_i \neq B_i$ then we obtain $\mathcal{M}atch(t,t',E,E') = \emptyset$.

Notice that we do not remove the trivial match-equations of the form $x \ll^?_\emptyset x$ from the system but we check the equality of the types of the variable x from the two sides in the corresponding contexts.

We can, of course, integrate the type constraints in the matching rules but keeping them separate allows us to use the same set of matching rules in the typed and untyped approaches.

The function $\mathcal{S}olution$ in the typed and syntactic case is defined by

$$\mathcal{S}olution(E \vdash t \ll^?_\emptyset E' \vdash t') = \mathcal{M}atch(t,t',E,E')$$

When the contexts E, E' of the terms t, t' are clear we omit them and we abbreviate the function $\mathcal{S}olution(E \vdash t \ll^?_\emptyset E' \vdash t')$ by $\mathcal{S}olution(t \ll^?_\emptyset t')$.

3.6 The evaluation rules of the typed ρ-calculus

The evaluation rules of the untyped $\rho_\emptyset$-calculus from [CK99a] enriched with the typing information are presented in Figure 3. The rules that are modified are the ones handling rewrite rules: *Fire* and *Switch*$_R$.

The typed $\rho_\mathcal{T}$-calculus (the ρ-calculus with a given a matching theory $\mathcal{T}$) is obtained similarly from the untyped $\rho_\mathcal{T}$-calculus but the typing system is more technically involved and is not considered in this paper. The type system dealing with rewrite rules with a left-hand side more elaborated than a first order term is slightly more complicated and is presented in [Cir00]. We conjecture that a similar approach can be used for the $\rho_\mathcal{T}$-calculus using an equational matching theory $\mathcal{T}$ that satisfies certain conditions.

$$
\begin{array}{lll}
\mathit{Fire} & [l_{[\![E]\!]} \to r](t) & \Longrightarrow \{\sigma r\} \\
& \quad \text{if } \textit{the rule is applied in context } F & \\
& \quad \text{where } \{\sigma\} = \mathcal{S}olution(E \vdash l \ll^{?}_{\emptyset} F \vdash t) & \\
& & \\
\mathit{Congruence} & [f(u_1,\ldots,u_n)](f(v_1,\ldots,v_n)) & \Longrightarrow \\
& \{f([u_1](v_1),\ldots,[u_n](v_n))\} & \\
& & \\
\mathit{Congruence\_fail} & [f(u_1,\ldots,u_n)](g(v_1,\ldots,v_m)) & \Longrightarrow \emptyset \\
& & \\
\mathit{Distrib} & [\{u_1,\ldots,u_n\}](v) & \Longrightarrow \\
& \{[u_1](v),\ldots,[u_n](v)\} & \\
& & \\
\mathit{Batch} & [v](\{u_1,\ldots,u_n\}) & \Longrightarrow \\
& \{[v](u_1),\ldots,[v](u_n)\} & \\
& & \\
\mathit{Switch}_R & u_{[\![E]\!]} \to \{v_1,\ldots,v_n\} & \Longrightarrow \\
& \{u_{[\![E]\!]} \to v_n,\ldots,u_{[\![E]\!]} \to v_n\} & \\
& & \\
\mathit{OpOnSet} & f(v_1,\ldots,\{u_1,\ldots,u_m\},\ldots,v_n) & \Longrightarrow \\
& \{f(v_1,\ldots,u_1,\ldots,v_n),\ldots,f(v_1,\ldots,u_m,\ldots,v_n)\} & \\
& & \\
\mathit{Flat} & \{u_1,\ldots,\{v_1,\ldots,v_n\},\ldots,u_m\} & \Longrightarrow \\
& \{u_1,\ldots,v_1,\ldots,v_n,\ldots,u_m\} &
\end{array}
$$

Fig. 3. The evaluation rules of the typed $\rho_{\mathcal{T}}$-calculus

As we have already mentioned, the interpretation of the function symbols is overloaded in the sense of having several ranks. If in the left-hand side of the rule *Congruence* the first f has the rank $f \in \mathcal{F}_{(A_1 \rightarrowtail B_1)\times\ldots\times(A_n \rightarrowtail B_n)\rightarrowtail(A \rightarrowtail B)}$ and the second one $f \in \mathcal{F}_{A_1\times\ldots\times A_n \rightarrowtail A}$ then the symbol f from the right-hand side of the evaluation rule has the rank $f \in \mathcal{F}_{B_1\times\ldots\times B_n \rightarrowtail B}$. The arguments of a term built using the first f are rewrite rules (or some other terms of compound type) that are applied to the arguments of a term built using the second f and the overloading of the symbol f is obviously needed in order to correctly type these applications.

The *Congruence* rules are redundant with respect to *Fire*. Indeed, one could notice that the application of a term $f(t_1,\dots,t_n)$ on another ρ-term t (i.e. $[f(t_1,\dots,t_n)](t)$) evaluates, using the evaluation rules *Congruence* and *Congruence_fail*, to the same term as the application of the ρ-rewrite rule $f(x_1,\dots,x_n) \to f([t_1](x_1),\dots,[t_n](x_n))$ on the term t (in a formal way, $[f(x_1,\dots,x_n) \to f([t_1](x_1),\dots,[t_n](x_n))](t)$) using the evaluation rule *Fire*. Therefore, the *Congruence* rules represent the η expansion of the ρ-calculus that would be defined by:

$$\mathit{Eta}\ f(t_1,\dots,t_n) \mapsto\!\!\!\!\rightarrow f(x_1,\dots,x_n) \to f([t_1](x_1),\dots,[t_n](x_n))$$

that applied in the particular case of a constant a leads to

$$a \Longrightarrow x \to [a](x).$$

There are mainly two properties that we want to prove for the typed calculus. First, we prove that the typed ρ-calculus preserves types under reduction, property usually called *subject reduction.* Second, we prove that in the typed ρ-calculus there are no infinite reductions. Comparing to the proofs from similar approaches, like the λ-calculus, we have to show that the handling of the non-determinism represented by sets of terms is done properly. In the ρ-calculus we have singletons corresponding to a deterministic result, like in the λ-calculus, but we also deal explicitly with the possible failure represented by $\emptyset$ and with non-deterministic results represented by sets with more than one element.

This latter property does not hold in the untyped calculus. In the untyped ρ-calculus there are terms that do not have a normal form:

Example 3.10 The term $\omega\omega = [x \to [x](x)](x \to [x](x))$ reduces successively to

$$\begin{array}{l}
[x \to [x](x)](x \to [x](x)) \Longrightarrow_\rho \\
\{x/(x \to [x](x))\}[x](x) \ \triangleq\ \{[x \to [x](x)](x \to [x](x))\} \Longrightarrow_\rho \\
\dots \\
\{\dots\{[x \to [x](x)](x \to [x](x))\}\dots\} \Longrightarrow_\rho \dots
\end{array}$$

In the typed ρ-calculus the types of the bound variables should be given explicitly and the corresponding ρ-term $[x_{[\![x:A]\!]} \to [x](x)](x_{[\![x:A']\!]} \to [x](x))$ is not well-typed independently of the type of the variable x from the contexts of the left-hand side of the rewrite rules. This follows immediately from the typing rule *App* that needs on one hand the type $B \rightarrowtail C$ for the first x of the application and on the other hand the type B for the second x. But this is obviously impossible since the two variables x from the right-hand side of the rule should have the type of the variable x from the left-hand side (A or A') in order to type the rewrite rule $x_{[\![x:A]\!]} \to [x](x)$ using the typing rule *Rule*.

Example 3.11 Let us consider the symbol *cons* used for constructing the lists and *nil* the symbol representing an empty set.

When building integer lists, $cons \in \mathcal{F}_{Int \times ListInt \rightarrowtail ListInt}$ and $nil \in ListInt$. But the same symbols can be used for building boolean lists and in this case $cons \in \mathcal{F}_{Bool \times ListBool \rightarrowtail ListBool}$ and $nil \in ListBool$. According to our supposition, we also have $cons \in \mathcal{F}_{(Int \rightarrowtail Bool) \times (ListInt \rightarrowtail ListBool) \rightarrowtail (ListInt \rightarrowtail ListBool)}$ and $nil \in ListInt \rightarrowtail ListBool$.

Using the typing rule *Op* we obtain $\emptyset \vdash cons(1, nil) : ListInt$ and similarly $\emptyset \vdash cons(True, nil) : ListBool$.

We can transform the integer lists into boolean lists by using the rewrite rules $0 \to False$, $1 \to True$, $2 \to True$, ... that, according to the typing rule *Rule*, have the type $Int \rightarrowtail Bool$ in the empty context. For example, the list $cons(1, nil)$ can be transformed by applying the term $cons(1 \to True, nil)$. In this latter term $cons \in \mathcal{F}_{(Int \rightarrowtail Bool) \times (ListInt \rightarrowtail ListBool) \rightarrowtail (ListInt \rightarrowtail ListBool)}$ and since $nil \in ListInt \rightarrowtail ListBool$, we have $\emptyset \vdash cons(1 \to True, nil) : ListInt \rightarrowtail ListBool$. We have thus obtained for $cons(1 \to True, nil)$ the type of a rewrite rule transforming an integer list into a boolean list.

According to the typing rule *App*, we can type in the empty context the application $[cons(1 \to True, nil)](cons(1, nil)) : ListBool$. This term eventually reduces to $\{cons(True, nil)\}$ that is of type $ListBool$ by the typing rules *Op* and *Set*.

4 Subject reduction

We show now that the typed ρ-calculus has the *subject reduction* property.

Theorem 4.1 *For all ρ-terms a and a', if $a \Longrightarrow_\rho a'$ and $E \vdash a : A$, then $E \vdash a' : A$.*

Proof (sketch): We inspect the evaluation rules of the typed ρ-calculus one by one and we show that the left-hand side and the right-hand side of each rule have the same type in a given context. For each evaluation rule of the form $lhs \Longrightarrow rhs$ we show that if $E \vdash lhs : A$ then $E \vdash rhs : A$.

Fire $\quad [l_{[\![F|_l]\!]} \to r](t) \mapsto\!\!\!\!\rightarrow \{\sigma r\}$
if *the rule is applied in context E*
where $\{\sigma\} = \mathcal{S}olution(F \vdash l \ll^?_\emptyset E \vdash t)$

Let A such that $E \vdash [l_{[\![F|_l]\!]} \to r](t) : A$. Using the typing rule *App* we infer that $E \vdash t : B$ and $E \vdash (l_{[\![F|_l]\!]} \to r) : B \rightarrowtail A$. By the typing rule *Rule* we have $F|_l \vdash l : B$ and $F|_l \cdot E \vdash r : A$ with $F|_l \cdot E$ consistent. If σ is the solution of $(l \ll^?_\emptyset t)$ then, according to the hypothesis on the matching, $E \vdash \sigma$ and $Dom(\sigma) = F|_l$. Since $F|_l \cdot E$ is consistent then $Dom(\sigma) \cdot E$ is consistent and by Lemma 3.7, $E \vdash \sigma r : A$. By the typing rule *Set*

$E \vdash \{\sigma r\} : A$.

If the matching $(l \ll^?_\emptyset t)$ fails then the right-hand side $\emptyset$ satisfies the property by the typing rule *Empty*.

Distrib $\quad [\{u_1, \dots, u_n\}](t) \twoheadrightarrow \{[u_1](t), \dots, [u_n](t)\}$

Let A such that $E \vdash [\{u_1, \dots, u_n\}](t) : A$. By the typing rule *App* we infer $E \vdash t : B$ and $E \vdash \{u_1, \dots, u_n\} : B \rightarrowtail A$. By the typing rule *Set* we have $E \vdash u_i : B \rightarrowtail A$, $i = 1, \dots, n$ and by applying the typing rule *App* n times we obtain $E \vdash [u_i](t) : A$, $i = 1, \dots, n$. Finally, the typing rule *Set* leads to

$E \vdash \{[u_1](t), \dots, [u_n](t)\} : A$.

Flat $\quad \{u_1, \dots, \{v_1, \dots, v_n\}, \dots, u_m\} \twoheadrightarrow \{u_1, \dots, v_1, \dots, v_n, \dots, u_m\}$

Let A such that $E \vdash \{u_1, \dots, \{v_1, \dots, v_n\}, \dots, u_m\} : A$. By the typing rule *Set* we have $E \vdash u_j : A$, $j = 1, \dots, m$, and $E \vdash \{v_1, \dots, v_n\} : A$. By the typing rule *Set* we obtain $E \vdash v_i : A$, $i = 1, \dots, n$ and finally, by the typing rule *Set* we have

$E \vdash \{u_1, \dots, v_1, \dots, v_n, \dots, u_m\} : A$.

The proof for the other rules is similar. $\square$

As we have pointed out in Section 3.6 the *Congruence* rules are redundant with respect to *Fire*. We would like to obtain a similar equivalence at the type level. Let us consider a function symbol f such that $f \in \mathcal{F}_{A_1 \times \dots \times A_n \rightarrowtail A}$, $f \in \mathcal{F}_{B_1 \times \dots \times B_n \rightarrowtail B}$, $f \in \mathcal{F}_{(A_1 \rightarrowtail B_1) \times \dots \times (A_n \rightarrowtail B_n) \rightarrowtail (A \rightarrowtail B)}$.

Then we can type the term $f(t_1, \dots, t_n)$ in a context E:

$$\frac{E \vdash t_1 : A_1 \rightarrowtail B_1 \ \dots \ E \vdash t_n : A_n \rightarrowtail B_n}{E \vdash f(t_1, \dots, t_n) : A \rightarrowtail B} \; Op$$

If we consider the term $f(x_1, \dots, x_n)_{[\![E']\!]} \to f([t_1](x_1), \dots, [t_n](x_n))$ with the context $E' = x_1 : A_1 \cdot \dots \cdot x_n : A_n$, then we obtain

$$\frac{E' \vdash x_1 : A_1 \ \dots \ E' \vdash x_n : A_n}{E' \vdash f(x_1, \dots, x_n) : A} \; Op,$$

$$\frac{E' \cdot E \vdash t_i : A_i \rightarrowtail B_i \quad E' \cdot E \vdash x_i : A_i}{E' \cdot E \vdash [t_i](x_i) : B_i} \; App,$$

$$\frac{E' \cdot E \vdash [t_i](x_i) : B_i}{E' \cdot E \vdash f([t_1](x_1), \dots, [t_n](x_n)) : B} \; Op$$

and by the typing rule *Rule*

$$\frac{E' \vdash f(x_1, \dots, x_n) : A \qquad E' \cdot E \vdash f([t_1](x_1), \dots, [t_n](x_n)) : B}{E \vdash f(x_1, \dots, x_n)_{[\![E']\!]} \to f([t_1](x_1), \dots, [t_n](x_n)) : A \rightarrowtail B}$$

Using the previous deductions we can conclude that

$$\frac{E' \vdash x_i : A_i \qquad E \vdash t_i : A_i \rightarrowtail B_i \qquad i = 1, \dots, n}{E \vdash f(x_1, \dots, x_n)_{[\![E']\!]} \to f([t_1](x_1), \dots, [t_n](x_n)) : A \rightarrowtail B}$$

We have thus induced the same type for a term $f(t_1, \dots, t_n)$ and for the corresponding extended term $f(x_1, \dots, x_n)_{[\![E']\!]} \to f([t_1](x_1), \dots, [t_n](x_n))$ in the same context E. One should notice that in the second term we should define explicitly in the context of the left-hand side of the rewrite rule the types of the bound variables of the rule.

This shows that our choice for the type system is reasonably good since consistent with the implicit extensionality rule.

5 Strong normalization

We concentrate now on the proof of the strong normalization of well-typed ρ-terms. This property guarantees the existence of a normal form and when the ρ-calculus is confluent we can conclude the uniqueness of normal forms.

Definition 5.1 A typed or untyped ρ-term t is *strongly normalizing (SN)* with respect to a reduction relation, iff all reductions starting at t are finite. The term is *weakly normalizing* iff it reduces to a normal form.

It is not surprising that, because of the relationship between the ρ-calculus and the λ-calculus, our proof of the strong normalization of the ρ-calculus is inspired from the proof of the strong normalization of the λ-calculus. There are several methods for proving the strong normalization of the λ-calculus. One is called *internalization* and was first used by Gandy [Gan80]. Another one is usually called *reducibility* and is based on the notions introduced by Tait [Tai67]. The latter technique has been generalized in [Gir72] and [JO97].

In what follows we use the notations, definitions and the proof line from [HS86] which is a variation of the Tait's method. Comparing to this method, in our approach we should handle properly the terms having a first order head symbol and the set terms.

When the context E in which we type the terms is clear we omit it and in this case we abbreviate $E \vdash t : A$ by $t : A$.

Definition 5.2 We define the *strong computability (SC)* of a term t by induction on the number of occurrences of type arrows "$\rightarrowtail$" in the type of t:

a. a term of atomic type is SC iff it is SN,

b. a term t such that $E \vdash t : A \rightarrowtail B$ is SC iff, for every SC term u such that $E \vdash u : A$, the term $[t](u)$ with $E \vdash [t](u) : B$ is SC.

The definition is extended for typed substitutions and we say that a substitution of the form $\{x_1 : A_1/u_1, \ldots, x_n : A_n/u_n\}$ is SC iff all terms u_i are SC.

The normalization proof is done in two steps. First we prove that every typeable term that is SC is SN. Second we show that all typeable terms are SC and we conclude that all typeable terms are SN.

Lemma 5.3 (every SC term is SN)

For all type A we have the following properties:

a. Given an atom t and the terms $u_1, \ldots, u_n$. If $u_1, \ldots, u_n$ are SN then $[\ldots[[t](u_1)](u_2)\ldots](u_n) : A$ is SC.

b. Every SC term of type A is SN.

Lemma 5.4 Given the contexts F and E and the terms l, r and t such that $F \vdash l : A$, $F|_l \cdot E \vdash r : B$ and $E \vdash t : A$. We consider the substitution σ such that $\{\sigma\} = \mathcal{S}olution(l \ll^?_\emptyset t)$.

If the terms σr and l, t are SC, then the term $[l_{[\![F|_l]\!]} \to r](t)$ is SC.

Lemma 5.5 (every typeable term is SC)

For every typed ρ-term t such that $E \vdash t : B$ we have:

a. t is SC,

b. For all SC substitutions σ_j, $j = 1, \ldots, m$ such that $E' \vdash \sigma_j$, the term $t^* = \sigma_1 \ldots \sigma_m t$ with $E' \vdash t^* : B$ is SC, where $E = \mathit{Dom}(\sigma_1) \cdot \ldots \cdot \mathit{Dom}(\sigma_m) \cdot E'$.

Proof (sketch): Part (a) is the special case of (b) when σ_i, $i = 1, \ldots, n$ are the identity substitution. We prove (b) by induction on the construction of t and the more elaborate case deals with a rewrite rule $t : B = t_{1[\![F|_{t_1}]\!]} \to t_2$.

We have $F \vdash t_1 : A$ and $F|_{t_1} \cdot E \vdash t_2 : C$ and $B = A \rightarrowtail C$. In this case $t^* = t_1^* \to t_2^*$ if we neglect changes in the bound variables. Due to the definition of substitution application, $t_1^* = t_1$.

We have to prove that for all SC terms u such that $E' \vdash u : A$, the term $[t^*](u)$ is SC or equivalently that $[t_1 \to t_2^*](u)$ is SC.

If the matching $(t_1 \ll^?_\emptyset u)$ fails then the result is $\emptyset$ and the property holds obviously.

We can easily prove that if $l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and syntactic matching is considered then, for any SC terms l, t and substitution $\{\mu\} = \mathcal{S}olution(l \ll^?_\emptyset t)$ we obtain that μ is SC.

We consider $\{\mu\} = \mathcal{S}olution(t_1 \ll^?_\emptyset u)$ and according to the hypothesis on the matching, $E' \vdash \mu$ and since t_1 and u are SC then it follows that μ is SC. By Lemma 3.7, $F|_{t_1} \cdot E' \vdash t_2^* : C$ and since $\mathit{Dom}(\mu) = F|_{t_1}$ then $E' \vdash \mu t_2^* : C$. By induction hypothesis applied with $m+1$ instead of m we obtain that μt_2^* is SC and thus, by Lemma 5.4, $[t^*](u)$ is SC. □

Theorem 5.6 *The typed ρ-calculus is strongly normalizing.*

Proof: The result follows immediately by the Lemma 5.5 and Lemma 5.3. □

6 Conclusion

We have shown that using the right notion of types and contexts, the simply typed $\rho_\emptyset$-calculus is type preserving and normalizing. Together with the confluence of this calculus, obtained under specific restrictions of the evaluation strategy [CK99b], this shows that despite its increased expressive capabilities this framework still preserves the main properties. This could be extended to

$\rho_{\mathcal{T}}$-calculus where the theory $\mathcal{T}$ is equational and has a decidable matching problem.

Our next goals are to study more elaborated type systems allowing us in particular object oriented as well as polymorphism features.

The untyped ρ-calculus has been used to give a semantics to the language ELAN and this approach can be smoothly extended to a representation of the ELAN typed rewrite rules and strategies [Bor98] in the typed ρ-calculus.

We have recently proposed a new presentation of the ρ-calculus [CKL00] allowing us to encode two major object-calculi [AC96,FHM94] in a very natural and simple way. This approach has been presented in an untyped context but we are also exploring a more elaborated and less restricitive type system than the one presented in this paper, allowing in particular to type self-applications.

References

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Verlag, 1996.

[BKK$^+$98] P. Borovanský, C. Kirchner, H. Kirchner, P. E. Moreau, and C. Ringeissen. An overview of elan. In C. Kirchner and H. Kirchner, editors, *Proceedings of the second International Workshop on Rewriting Logic and Applications*, volume 15, Pont-à-Mousson (France), September 1998. Electronic Notes in Theoretical Computer Science.

[Bor98] P. Borovanský. *Le contrôle de la réécriture: étude et implantation d'un formalisme de stratégies*. Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, France, October 1998.

[CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[Cir00] H. Cirstea. *Le Rho Calcul: Fondements et Applications*. PhD thesis, Universié Henri Poincaré - Nancy I, 2000. to appear.

[CK99a] H. Cirstea and C. Kirchner. Combining higher-order and first-order computation using ρ-calculus: Towards a semantics of ELAN. In D. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, Research Studies, ISBN 0863802524, pages 95–120. Wiley, 1999.

[CK99b] H. Cirstea and C. Kirchner. An introduction to the rewriting calculus. Research Report RR-3818, INRIA, December 1999.

[CKL00] H. Cirstea, C. Kirchner, and L. Liquori. Matching power. July 2000. Submitted.

[Deu96] A. Deursen. An Overview of ASF+SDF. In *Language Prototyping*, pages 1–31. World Scientific, 1996. ISBN 981-02-2732-9.

[DHK00] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. *Information and Computation*, 157(1/2):183–235, 2000.

[FHM94] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specializatio n. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[FN97] K. Futatsugi and A. Nakagawa. An overview of CAFE specification environment – an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In *Proceedings of the 1st IEEE Int. Conference on Formal Engineering Methods*, 1997.

[Gan80] R. Gandy. Proof of strong normalisation. In J. P. Seldin and J. R. Hindley, editors, *Essays on Combinatory Logic, Lambda Calculus, and Formalism.* Academic Press inc., New York (NY, USA), 1980.

[GBT89] J. Gallier and V. Breazu-Tannen. Polymorphic rewriting conserves algebraic strong normalization and confluence. In *16th Colloquium Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 137–150. Springer-Verlag, 1989.

[Gir72] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, June 1972.

[Hin97] J. R. Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University, 1997.

[HS86] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and Lambda-calculus.* Cambridge University, 1986.

[Hue76] G. Huet. *Résolution d'equations dans les langages d'ordre 1,2, ...,ω.* Thèse de Doctorat d'Etat, Université de Paris 7 (France), 1976.

[JO97] J.-P. Jouannaud and M. Okada. Abstract data type systems. *Theoretical Computer Science*, 173(2):349–391, 1997.

[KKV93] C. Kirchner, H. Kirchner, and M. Vittek. Implementing computational systems with constraints. In P. Kanellakis, J.-L. Lassez, and V. Saraswat, editors, *Proceedings of the first Workshop on Principles and Practice of Constraint Programming, Providence (R.I., USA)*, pages 166–175. Brown University, 1993.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[Mil84] R. Milner. A proposal for standard ML. In *Proceedings ACM Conference on LISP and Functional Programming*, 1984.

[Oka89] M. Okada. Strong normalizability for the combined system of the typed Lambda-calculus and an arbitrary convergent term rewrite system. In

Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation, Portland (Oregon), pages 357–363. ACM Press, July 1989. Report CRIN 89-R-220.

[PJ87] S. Peyton-Jones. *The implementation of functional programming languages.* Prentice Hall, Inc., 1987.

[Tai67] W. Tait. Intensional interpretation of functionals of finite type I. *The Journal of Symbolic Logic*, 32, 1967.

[Vis99] E. Visser. Strategic pattern matching. In P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30–44, Trento, Italy, July 1999. Springer-Verlag.