Decidable Type Inference for the Polymorphic Rewriting Calculus

Horatiu Cirstea[1] & Claude Kirchner[2] & Luigi Liquori[3] & Benjamin Wack[4]

*1: University of Nancy II & LORIA** Horatiu.Cirstea@loria.fr
*2: INRIA & LORIA** Claude.Kirchner@loria.fr
3: INRIA Sophia Antipolis Luigi.Liquori@inria.fr
*4: University Henri Poincaré & LORIA** Benjamin.Wack@loria.fr

Résumé

The rewriting calculus (ρ-calculus), is a minimal framework embedding λ-calculus and Term Rewriting Systems, that allows abstraction on variables and patterns. The ρ-calculus features higher-order functions (from λ-calculus) and pattern matching (from Term Rewriting Systems). In this paper, we study extensively the decidability of type inference in the second-order ρ-calculus *à la* Curry (rhoStk).

1. Introduction

A promising line of research unifying the logic paradigm with the functional paradigm is that of *rewrite-based* languages [MOM02] like, for example, ELAN [The04d], Maude [The04c], ASF+SDF [The04a], Obj* [FN96, Gog04]). This kind of languages have been successfully used for theorem proving [BCM04], constraint solving [Cas96], model-checking [EMS03, CMR04], program transformation [vdBKV96], etc. Moreover, rewrite-based programming can be integrated in other languages [MRV03] as "formal islands" (that we can reason on) in the middle of imperative programs [MRV03, KMR05].

The main mechanism the rewrite-based languages are based on is the *pattern-matching* which allows one to discriminate between alternatives. Each pattern is associated with an action ; once an instance of a pattern is recognized, the corresponding term is rewritten to a new one.

Useful applications of pattern-matching lie in the field of pattern recognition, and strings/trees manipulation. It has also been widely used in functional and logic programming, for instance in ML [MTHM97, The03a], Haskell [The04b], Scheme [The04e], or Prolog [The03b]. However, in all these applications, pattern-matching is considered as a convenient mechanism for expressing complex requirements about the function's argument, rather than a basis for a paradigm of computation. We argue that the computational behavior of a calculus can be deeply influenced by the presence of pattern-matching and we support this statement by studying the (typing related) properties of a calculus that strongly relies on the matching mechanism.

One of the most commonly used models of computation, the lambda calculus, uses only trivial pattern-matching. This calculus has recently been extended, initially for programming concerns, either by introducing patterns in Lambda-calculi [Pey87, vO90], or by introducing matching and rewrite rules in functional languages. More concerned with extending logics, Stehr has studied a Calculus of Constructions enhanced with rewriting logic [Ste02].

The *rewriting calculus* [CKL01b, CLW04] is a foundational framework integrating matching, rew-

[0]LORIA: UMR 7506, CNRS, INRIA, INPL, UHP, Université de Nancy II.

riting and functions in a uniform way. Its abstraction mechanism is based on the rewrite rule formation : in a term of the form $P \twoheadrightarrow A$, one abstracts over the (free variables of the) pattern P.

If an abstraction $P \twoheadrightarrow A$ is applied to the term B, then the evaluation mechanism is based on the instantiation (in A) of the free variables present in P with the appropriate subterms of B. Indeed, this instantiation is achieved by matching P against B.

As a foundational calculus, the rewriting calculus is a non-trivial generalization of the lambda calculus, since we get the lambda calculus back if every pattern P is a variable. Term rewrite systems can also be conveniently modeled in the rewriting calculus [CLW04] by using the *structure* operator for representing the corresponding sets of rewrite rules as ρ-terms. In particular, the notions of *rule application* and *result* (basic ingredients of term rewrite systems) become explicit in the rewriting calculus.

In the rewriting calculus, a rewrite rule is a first-class citizen, which can be created, manipulated and modified during the evaluation. The abilities to manipulate rules and to define strategies guiding their application represent the basic methods in rewrite-based languages and thus the rewriting calculus can be used as a core engine calculus for this kind of languages.

It is well known that static analysis via a type system enforces a safer programming discipline. We present and analyze here a powerful polymorphic type system for the rewriting calculus that can be seen as a good candidate for giving the static semantics of a family of rewrite-based languages such as ELAN and Maude.

In [LW05] we have introduced a ρ-calculus *à la* Church (called RhoF) featuring second-order polymorphic types. In this fully typed second-order rewriting calculus, the types of the bound variables are specified in the term, making type reconstruction and verification quite straightforward. Moreover, this calculus enjoys classical type related properties such as subject reduction, and type uniqueness.

We have also proposed a classical erasing function [Cur34, Lei83, GR88] that can be applied to RhoF in order to obtain a corresponding type inference system *à la* Curry. In Rho|F|, the calculus *à la* Curry, type information is not given in the term, and the type system is not fully syntax-directed, thus enforcing a flexible polymorphic type discipline. When we look at the ρ-calculus as a kernel calculus underneath a pattern-matching based programming language, this approach corresponds to ELAN, or Maude, or Obj*, or ASF+SDF, or Haskell, or ML-like languages, where the user can write programs in a completely untyped language, and types are automatically inferred at compilation-time. Type inference can be also intended as the construction of an abstract interpretation of the program, that can be used as a correctness criterion. Unfortunately, as it is well-known for the λ-calculus [Wel99], the type assignment problem for Rho|F| is undecidable.

We introduce in what follows a restriction of Rho|F|, called rhoStk, where the polymorphic types are clearly separated from the polymorphic type schemes. We discuss its expressive power and we present a type inference algorithm. We compare our approach to similar ones used in functional programming languages and more precisely in ML.

Synopsis In the next section we introduce the syntax and semantics of rhoStk together with a relation defining in some sense the definitive matching failures that one wants normally to eliminate from the final result of an evaluation. We introduce then the typing system and the type inference algorithm that is proved sound, correct and principal. In Section 3 we briefly compare the approach presented here with the one used in ML. The final section concludes and gives some possible directions for future research. The complete proofs of the properties stated in the paper are available in [Wac05, CKLW06].

K	$\in$	$\mathcal{K}$	K	$::=$	$*$
$\tau.\iota$	$\in$	$\mathcal{T}\!ype$	τ	$::=$	$\alpha \mid \iota_{\overline{\alpha}} \mid \tau \multimap \tau$
σ	$\in$	$\mathcal{T}\!ype\mathcal{S}cheme$	σ	$::=$	$\forall\overline{\alpha}.\tau$
$\Gamma.\Delta$	$\in$	$\mathcal{C}ontext$	Δ	$::=$	$\emptyset \mid \Delta, f{:}\sigma \mid \Delta, X{:}\sigma$
$P.Q$	$\in$	$\mathcal{P}$	P	$::=$	$\mathsf{stk} \mid X \mid f(\overline{P})$ *(all vars occur only once in any P)*
$A.B.f$	$\in$	$\mathcal{T}$	A	$::=$	$\mathsf{stk} \mid f \mid X \mid P \multimap A \mid \mathsf{let}\ P \ll A\ \mathsf{in}\ A \mid A\,A \mid A \wr A$

FIG. 1 – Syntax of rhoStk

2. Type inference in rhoStk

We now define rhoStk, a polymorphic rewriting calculus *à la* Curry where type information is not given in the term. In order to recover the decidability of the type inference, polymorphic types are clearly separated from the polymorphic type schemes. We discuss the expressive power of the introduced calculus and we present a type inference algorithm.

2.1. Syntax

We consider the meta-symbols "$\_ \multimap \_$" (function- and type-abstraction), and "$\mathsf{let}\ \_ \ll \_\ \mathsf{in}\ \_$" (delayed matching constraint), and "$\_ \wr \_$" (structure operator). The application operator is denoted by concatenation. 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 "$\mathsf{let}\ \_ \ll \_\ \mathsf{in}\ \_$" which is higher than that of "$\_ \multimap \_$" which is, in turn, of higher priority than the "$\_ \wr \_$". The symbol τ ranges over the set $\mathcal{T}\!ype$ of types, the symbol ι ranges over the set $\mathcal{T}\!ype_K$ of type constants, the symbols α, β range over the set $\mathcal{T}\!ype_V$ of type-variables, the symbol σ ranges over the set $\mathcal{T}\!ype\mathcal{S}cheme$ of type schemes, the symbols $A, B, C, \ldots, U, V, W$ range over the set $\mathcal{T}$ of (un)typed terms, the symbols $X, Y, Z, \ldots$ range over the set $\mathcal{V}$ of term variables, the symbols $a, b, c, \ldots, f, g, h, \ldots$ range over a set $\mathcal{T}\!erm_K$ of term constants. The symbols P, Q range over the set $\mathcal{P}$ of patterns and the symbols θ, ϕ, ψ range over substitutions. We denote $\overline{A}$ for $A_1 \cdots A_n$, for $n \geq 0$. The application of a constant, say f, to a term A will be usually denoted by $f(A)$, following the algebraic folklore ; this convention can be curryfied in order to denote a function taking multiple arguments, *e.g.* $f(\overline{A}) \triangleq f(A_1, \cdots, A_n) \triangleq (\cdots (f\,A_1) \cdots A_n)$.

The syntax of rhoStk is presented in FIGURE 1. As one would expect, the *types* allow us to define a polymorphic type system (*i.e.* type-variables can be bound in types through the "$\forall$" binder). The *patterns* are algebraic terms (*i.e.* terms constructed only with variables, constants and applications) 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 [vO90] is needed to keep the small-step semantics confluent. A *rewrite rule* of the form $P \multimap A$ abstracting over the variables of P is a first-class citizen of the calculus. An *application* is implicitly denoted by concatenation. The *delayed matching constraint* $\mathsf{let}\ P \ll A\ \mathsf{in}\ B$ can be seen as the term B with its free-variables constrained by the matching between P and A. The symbol stk is the special constant representing all the definitive (matching) failures.A *structure* is a collection of terms that can be seen either as a set of rewrite rules or as a set of results. The variables of the left-hand side of a rewrite rule are bound in a usual way.

Definition 2.1 (Free-variables $\mathcal{FV}$)

$$
\begin{array}{lcllcl}
\mathcal{FV}(f) & \triangleq & \emptyset & \mathcal{FV}(P \twoheadrightarrow A) & \triangleq & \mathcal{FV}(A) \setminus \mathcal{FV}(P) \\
\mathcal{FV}(\mathsf{stk}) & \triangleq & \emptyset & \mathcal{FV}(\mathsf{let}\ P \ll A\ \mathsf{in}\ B) & \triangleq & \mathcal{FV}((P \twoheadrightarrow B)\, A) \\
\mathcal{FV}(X) & \triangleq & \{X\} & \mathcal{FV}(A\, B) & \triangleq & \mathcal{FV}(A) \cup \mathcal{FV}(B) \\
\mathcal{FV}(\alpha) & \triangleq & \{\alpha\} & \mathcal{FV}(A \wr B) & \triangleq & \mathcal{FV}(A) \cup \mathcal{FV}(B) \\
 & & & \mathcal{FV}(\tau_1 \twoheadrightarrow \tau_2) & \triangleq & \mathcal{FV}(\tau_1) \cup \mathcal{FV}(\tau_2)
\end{array}
$$

As usual, we work modulo *α-conversion* and we adopt Barendregt's *"hygiene-convention"* [Bar84], *i.e.* free- and bound-variables have different names. Since these assumptions avoid the variable capture problems, the definition of substitution applications is straightforward.

Definition 2.2 (Substitutions) *A substitution θ is a mapping from the set of term variables (resp. type variables) to the set of terms (resp. types). A finite substitution θ has the form $\{A_1/X_1 \dots A_m/X_m\}$, or $\{\tau_1/\alpha_1 \dots \tau_m/\alpha_m\}$, and its domain $\mathcal{D}om(\theta)$ denotes $\{X_1, \dots, X_m\}$, resp. $\{\alpha_1, \dots, \alpha_m\}$. The application of a substitution θ to a term A (resp. type τ, resp. context Δ), denoted by $A\theta$ (resp. $\tau\theta$, resp. $\Delta\theta$), is defined as follows :*

$$
\begin{array}{lcllcl}
X_i\theta & \triangleq & \begin{cases} A_i & \text{if} X_i \in \mathcal{D}om(\theta) \\ X_i & \text{otherwise} \end{cases} & \alpha_i\theta & \triangleq & \begin{cases} \tau_i & \text{if } \alpha_i \in \mathcal{D}om(\theta) \\ \alpha_i & \text{otherwise} \end{cases} \\
f\theta & \triangleq & f & \iota\theta & \triangleq & \iota \\
\mathsf{stk}\theta & \triangleq & \mathsf{stk} & \emptyset\theta & \triangleq & \emptyset \\
(P \twoheadrightarrow A)\theta & \triangleq & P \twoheadrightarrow A\theta & (\tau_1 \twoheadrightarrow \tau_2)\theta & \triangleq & \tau_1\theta \twoheadrightarrow \tau_2\theta \\
(A\, B)\theta & \triangleq & A\theta\, B\theta & (\Delta, X{:}\tau)\theta & \triangleq & \Delta\theta, X{:}\tau\theta \\
(A \wr B)\theta & \triangleq & A\theta \wr B\theta & (\Delta, f{:}\tau)\theta & \triangleq & \Delta\theta, f{:}\tau\theta \\
(\mathsf{let}\ P \ll A\ \mathsf{in}\ B)\theta & \triangleq & (\mathsf{let}\ P \ll A\theta\ \mathsf{in}\ B\theta) & & &
\end{array}
$$

2.2. Semantics

The evaluation mechanism of the calculus relies on the fundamental operation of *matching* that allows us to bind variables to their current values. Since we want to define an expressive and powerful calculus, we allow the matching to be performed *modulo* a congruence on terms. This congruence used at matching time is a fundamental parameter of the calculus and different instances are obtained when instantiating this parameter by a congruence defined, for example, syntactically, or equationally or in a more elaborated way [CKL01a].

For the purpose of this paper we restrict to syntactic matching and we say that a substitution θ is solution of the matching-equation $A \prec\!\!\!\ll B$ if $A\theta \equiv B$, *i.e.*$A\theta$ and B and identical. The unique solution of such a matching problem is denoted $\theta_{A \prec\!\!\ll B}$.

It is sometimes interesting to handle uniformly the (definitive) matching failures and to eliminate them when not significant for the computation. We want thus to represent by stk all the delayed matching constraints whose corresponding matching problem is unsolvable independently of subsequent instantiations and reductions, and thus we intuitively want to define stk by the rule :

$$
\frac{\nexists\theta,\ P\theta =_{\rho\sigma\delta} N\theta}{\mathsf{let}\ P \ll N\ \mathsf{in}\ M \to_{\mathsf{stk}} \mathsf{stk}}
$$

where $=_{\rho\sigma\delta}$ is the congruence induced by the first three evaluation rules given in FIGURE 2. The conditions of this reduction rule are undecidable but we can define a sufficient condition guaranteeing that a given term will never match a given pattern.

$$
\begin{array}{rcll}
(P \twoheadrightarrow A)\, B & \to_\rho & \mathsf{let}\ P \ll B\ \mathsf{in}\ A & \\
\mathsf{let}\ P \ll B\ \mathsf{in}\ A & \to_\sigma & A\theta_{(P \prec\!\!\!\ll B)} & \textit{Provided } \theta \textit{ exists} \\
(A \wr B)\, C & \to_\delta & A\,C \wr B\,C & \\
A & \to_{\mathsf{stk}} & B & \textit{As defined in Definition 2.4}
\end{array}
$$

FIG. 2 – Top-level Rules of rhoStk

Definition 2.3 (Superposition) *The relation $\not\sqsubseteq$ is defined on $\mathcal{P} \times \mathcal{T}$ as follows :*

$$
\begin{array}{rcllrcl}
\mathsf{stk} & \not\sqsubseteq & g(N_1, \ldots, N_n) & & f(P_1, \ldots, P_m) & \not\sqsubseteq & \mathsf{stk} \\
\mathsf{stk} & \not\sqsubseteq & P \twoheadrightarrow N & & f(P_1, \ldots, P_m) & \not\sqsubseteq & P \twoheadrightarrow N \\
f(P_1, \ldots, P_m) & \not\sqsubseteq & g(N_1, \ldots, N_n) & & \multicolumn{3}{l}{\textit{if } f \not\equiv g \textit{ or } n \neq m \textit{ or } \exists i,\ P_i \not\sqsubseteq N_i} \\
f(P_1, \ldots, P_m) & \not\sqsubseteq & \mathsf{let}\ P \ll N\ \mathsf{in}\ M & & \multicolumn{3}{l}{\textit{if } P \not\sqsubseteq N \textit{ or } f(P_1, \ldots, P_m) \not\sqsubseteq M}
\end{array}
$$

Lemma 2.1 (Correction of $\not\sqsubseteq$) *For any P and M, if $P \not\sqsubseteq M$ then*

$$\forall \theta_1, \theta_2,\ \forall M',\ M\theta_1 \mapsto\!\!\!\!\to^{\mathsf{stk}}_{\rho\sigma\delta} M' \ \Rightarrow\ P\theta_2 \not\equiv M'$$

Corollary 2.1 (Stability of $\not\sqsubseteq$) *The relation $P \not\sqsubseteq M$ is stable by substitution and reduction of M.*

Starting from the superposition relation, we define a reduction relation that eliminates definitively stuck subterms.

Definition 2.4 *The relation $\to_{\mathsf{stk}}$ is defined by the following rules :*

$$
\begin{array}{lcll}
\mathsf{let}\ P \ll A\ \mathsf{in}\ B & \to_{\mathsf{stk}} & \mathsf{stk} & \mathsf{if}\ P \not\sqsubseteq A \\
\mathsf{stk} \wr A & \to_{\mathsf{stk}} & A & \\
A \wr \mathsf{stk} & \to_{\mathsf{stk}} & A & \\
\mathsf{stk}\, A & \to_{\mathsf{stk}} & \mathsf{stk} &
\end{array}
$$

We denote by $\mapsto_{\mathsf{stk}}$ the contextual closure induced by these rules. Its reflexive and transitive closure is denoted by $\mapsto\!\!\!\!\to_{\mathsf{stk}}$.

FIGURE 2 shows the reduction rules of rhoStk :

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

(σ) this rule applies if and only if the matching equation $P \prec\!\!\!\ll B$ has a solution : in this case the matching solution is computed and applied to the term A. If there is no solution, this rule does not apply. As we shall see, further reductions or instantiations are likely to modify B so that the equation has a solution and the rule can be triggered.

(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.

(stk) pushes into the operational semantics the rewrite rules dealing with the elimination of the definitive failures.

We denote by $\mapsto_{\rho\sigma\delta}$ the contextual closure induced by the rules (ρ), (δ) and (σ). Its reflexive and transitive closure is denoted by $\mapsto\!\!\!\!\to_{\rho\sigma\delta}$. The symmetric and transitive closure of $\mapsto\!\!\!\!\to_{\rho\sigma\delta}$ is denoted by $=_{\rho\sigma\delta}$. We denote by $\mapsto^{\mathsf{stk}}_{\rho\sigma\delta}$ the relation $\mapsto_{\mathsf{stk}} \cup \mapsto_{\rho\sigma\delta}$ and by $\mapsto\!\!\!\!\to^{\mathsf{stk}}_{\rho\sigma\delta}$ its reflexive and transitive closure.

Theorem 2.1 (Confluence of rhoStk [CLW04, Wac05]) *The relation* $\mapsto^{\mathsf{stk}}_{\rho\sigma\delta}$ *is confluent.*

Example 2.1 (Computing the length of a list) *Let use define the* ρ*-term*

$$len \;\triangleq\; rec\; S \twoheadrightarrow \left(\begin{array}{ll} nil & \twoheadrightarrow 0 \\ \wr & \\ (cons\, X\, L) & \twoheadrightarrow suc\,(S\,(rec\; S)\, L) \end{array}\right)$$

where $rec, nil, 0, cons, suc$ *are constants. Then the* ρ*-term* $len\,(rec\; len)$ *computes the length of any list that is given as an argument. We should mention that if the evaluation is performed using only* $\mapsto\!\!\!\!\rightarrow_{\rho\sigma\delta}$ *then the final result is a structure containing the expected result and several delayed matching constraints in normal form. When* $\mapsto\!\!\!\!\rightarrow^{\mathsf{stk}}_{\rho\sigma\delta}$ *is used, the final result consists only off the expected term.*

It is interesting to see that, if we erase all the occurrences of S *and* rec*, we get the classical rewrite system computing the length of lists.*

2.3. Typing Rules

As we have already mentioned, the type assignment system for Rho|F| is undecidable. One of the problems that is peculiar to this calculus is the ability to define any number of constants with a given type, without really considering them as constructors. Thus, in Rho|F|, a constant can have a type $f : \forall\alpha.\ (\alpha \to \iota)$ where the parameter α does not appear explicitly in the rightmost type ι. Then when typing

$$\mathsf{let}\;\; f(X) \ll f(Y \twoheadrightarrow Y) \;\;\mathsf{in}\;\; (X\;\, 1)$$

the pattern $f(X)$ gets type ι, where the type of $Y \twoheadrightarrow Y$ is forgotten. Then it is impossible to infer correctly the type of X : a standard algorithm would suggest the most general type $\forall\beta.\ (int \to \beta)$, so the type computed for the expression above can be anything.

This typing discipline for constants is unsound : the previous term has type β (for any β) but it reduces to 1, which has type *int*. Moreover, it leads to undecidability of typing. The inference algorithm could be easily patched to deal with the example above, but the problem is that the pattern could be replaced by a variable Z and the matching against $f(X)$ can then be arbitrarily nested in the body of the delayed matching constraint. Thus, we need to enrich the rightmost term of the type of f with all the type variables appearing in the whole type.

Moreover, a vast amount of types is available in Rho|F| since quantification can occur anywhere in a type. Therefore, we need to restrict polymorphism to well known "type schemes" of the form $\forall\overline{\alpha}.\tau$, where τ is a first-order type, *i.e.* a monomorphic-type. As example, $\forall\alpha.\alpha \twoheadrightarrow \alpha \;\simeq\; \{\tau \twoheadrightarrow \tau \mid \tau \in \mathcal{T}ype\}$ is the type-scheme for polymorphic identity. Type schemes are equivalent modulo α-conversion. We define simultaneous instantiations of type schemes, via a relation (denoted by $\leq$) as follows :

$$\tau_1 \leq \tau_2 \text{ iff } \tau_2 \triangleq \forall\overline{\alpha}.\tau_3 \text{ and } \tau_1 \triangleq \tau_3\{\overline{\tau}/\overline{\alpha}\} \text{ for suitable } \overline{\tau}.$$

The resulting type system is given in FIGURE 3 and proves judgment of the shape :

$$\Gamma \vdash ok \;\;\text{ and }\;\; \Gamma \vdash \tau : * \;\;\text{ and }\;\; \Gamma \vdash U : \tau$$

We briefly comment some important points concerning the typing rules :

- Formation of admissible type schemes follow some strict rules : every bound variable has to appear in the rightmost type, hence the side condition $\alpha \in \mathsf{Lab}(\sigma)$;
- $(Term{\cdot}Var)$, and $(Term{\cdot}Const)$: the type of a variable/constant is a type instance of its type-scheme ;

Well-formed Contexts

$$\frac{}{\emptyset \vdash ok}\ (Ctx\cdot Empty) \qquad \frac{\Gamma \vdash ok \quad X \notin \mathcal{D}om(\Gamma)}{\Gamma, X{:}\sigma \vdash ok}\ (Ctx\cdot Var)$$

$$\frac{\Gamma \vdash ok \quad \iota \notin \mathcal{D}om(\Gamma)}{\Gamma, \iota{:}* \vdash ok}\ (Ctx\cdot TypeConst) \qquad \frac{\Gamma \vdash ok \quad \Gamma \vdash \sigma : * \quad f \notin \mathcal{D}om(\Gamma)}{\Gamma, f{:}\sigma \vdash ok}\ (Ctx\cdot Const)$$

$$\frac{\Gamma \vdash ok \quad \alpha \notin \mathcal{D}om(\Gamma)}{\Gamma, \alpha{:}* \vdash ok}\ (Ctx\cdot Var^{\forall})$$

Well-kinded Type (Schemes)

$$\frac{\alpha \in \mathsf{Lab}(\sigma) \quad \Gamma, \alpha{:}* \vdash \sigma : *}{\Gamma \vdash \forall\alpha.\sigma : *}\ (TypeScheme^{\forall}) \quad \frac{\Gamma \vdash ok}{\Gamma \vdash \tau : *}\ (Type)$$

$\mathsf{Lab}(\iota_{\overline{\alpha}}) = \overline{\alpha}$ $\qquad$ $\mathsf{Lab}(\tau_1 \twoheadrightarrow \tau_2) = \mathsf{Lab}(\tau_2)$

$\mathsf{Lab}(\alpha) = \{\alpha\}$ $\qquad$ $\mathsf{Lab}(\forall\alpha.\sigma) = \mathsf{Lab}(\sigma)$

Well-formed Terms and Patterns

$$\frac{\Gamma_1, X{:}\sigma, \Gamma_2 \vdash ok \quad \sigma \leq \tau}{\Gamma_1, X{:}\sigma, \Gamma_2 \vdash X : \tau}\ (Term\cdot Var) \qquad \frac{\Gamma_1, f{:}\sigma, \Gamma_2 \vdash ok \quad \sigma \leq \tau}{\Gamma_1, f{:}\sigma, \Gamma_2 \vdash f : \tau}\ (Term\cdot Const)$$

$$\frac{\begin{array}{ll}\Gamma, \Delta \vdash P : \tau_1 & \mathcal{BV}(\mathcal{C}o\mathcal{D}om\Delta) = \emptyset \\ \Gamma, \Delta \vdash U : \tau_2 & \mathcal{D}om(\Delta) = \mathcal{FV}(P)\end{array}}{\Gamma \vdash P \twoheadrightarrow U : \tau_1 \twoheadrightarrow \tau_2}\ (Term\cdot Abst^{\twoheadrightarrow}) \qquad \frac{\begin{array}{c}\Gamma \vdash U : \tau_1 \twoheadrightarrow \tau_2 \\ \Gamma \vdash V : \tau_1\end{array}}{\Gamma \vdash U\, V : \tau_2}\ (Term\cdot Appl^{\twoheadrightarrow})$$

$$\frac{\begin{array}{ll}\Gamma \vdash V : \tau_1 & \\ \Gamma, \Delta \vdash P : \tau_1 & \mathcal{BV}(\mathcal{C}o\mathcal{D}om\Delta) = \emptyset \\ \Gamma, \mathsf{Gen}(\Delta;\Gamma) \vdash U : \tau_2 & \mathcal{D}om(\Delta) = \mathcal{FV}(P)\end{array}}{\Gamma \vdash \mathsf{let}\ P \ll V\ \mathsf{in}\ U : \tau_2}\ (Term\cdot Match) \qquad \frac{\begin{array}{c}\Gamma \vdash U : \tau \\ \Gamma \vdash V : \tau\end{array}}{\Gamma \vdash U \wr V : \tau}\ (Term\cdot Struct)$$

$\mathsf{Gen}(\tau;\Gamma) \quad \triangleq \quad \forall\overline{\alpha}.\tau$ where $\overline{\alpha} = \mathcal{FV}(\tau) \setminus \mathcal{FV}(\Gamma)$ and Gen is extended to contexts.

FIG. 3 – Terms of rhoStk

- $(Term\cdot Abst^{\twoheadrightarrow})$: the context Δ has to be inferred, but it can assign only types (not type-schemes) to the variables of P. It corresponds to the behavior of the typing rule for functional abstraction `fun` $x \to a$ in ML.
- $(Term\cdot Match)$: this rule performs a restricted form of polymorphic type inference. Again the context Δ used to type P assigns only types to the free variables of P, but when typing U the corresponding type-schemes can be used. It is an enhanced version of the ML let featuring matching.

Lemma 2.2 (Subject Reduction for rhoStk)
If $\Gamma \vdash U : \tau$ *and* $U \mapsto^{\mathsf{stk}}_{\rho\delta} V$*, then* $\Gamma \vdash V : \tau$.

Example 2.2 presents a simple type derivation in rhoStk for the problematic term shown at the beginning of this section.

Example 2.2 (A Simple Type Derivation in rhoStk)

$$
\dfrac{
(*)\;\dfrac{\dfrac{\dfrac{\beta \leq \beta}{\Gamma, Y{:}\beta \vdash Y : \beta}}{\Gamma \vdash Y \rightarrowtriangle Y : \beta \rightarrowtriangle \beta}}{\Gamma \vdash f(Y \rightarrowtriangle Y) : \kappa_{\beta \rightarrowtriangle \beta}}
\quad
\dfrac{(*)\;\dfrac{\beta \rightarrowtriangle \beta \leq \beta \rightarrowtriangle \beta}{\Gamma, X : \beta \rightarrowtriangle \beta \vdash X : \beta \rightarrowtriangle \beta}}{\Gamma, X : \beta \rightarrowtriangle \beta \vdash f(X) : \kappa_{\beta \rightarrowtriangle \beta}}
\quad
\dfrac{\dfrac{\iota \rightarrowtriangle \iota \leq \forall\beta.(\beta \rightarrowtriangle \beta)}{\Gamma, \Delta \vdash X : \iota \rightarrowtriangle \iota}\quad \dfrac{\iota \leq \iota}{\Gamma, \Delta \vdash 1 : \iota}}{\Gamma, \Delta \vdash X\,1 : \iota}
}{\Gamma \vdash \mathsf{let}\;\; f(X) \ll f(Y \rightarrowtriangle Y)\;\; \mathsf{in}\;\; (X\,1) : \iota}
$$

where $\Gamma \triangleq 1{:}\iota, f{:}\forall\alpha.(\alpha \rightarrowtriangle \kappa_\alpha)$, *and* $\Delta \triangleq X{:}\forall\beta.(\beta \rightarrowtriangle \beta)$ *and* $(*)$ *is* $\dfrac{(\beta \rightarrowtriangle \beta) \rightarrowtriangle \kappa_{\beta \rightarrowtriangle \beta} \leq \forall\alpha.(\alpha \rightarrowtriangle \kappa_\alpha)}{\Gamma \vdash f : (\beta \rightarrowtriangle \beta) \rightarrowtriangle \kappa_{\beta \rightarrowtriangle \beta}}$

We see that the pattern $f(X)$ is assigned a type $\kappa_{\beta \rightarrowtriangle \beta}$, ensuring that X has type $\beta \rightarrowtriangle \beta$. Then generalization gives it type $\forall\beta.(\beta \rightarrowtriangle \beta)$ when typing the body, which ensures that any type of x is an instance of $\beta \rightarrowtriangle \beta$.

The next section presents a type inference algorithm (called $\mathsf{W}^{\ll}$) that gives a solution to this problem.

Example 2.3 (Typing the term $len(rec\;\; len)$)

Let us consider again the term from Example 2.1. If we consider the context

$$
\Gamma \;\;=\;\; \left\{ \begin{array}{l} 0 : int, suc : int \rightarrowtriangle int, cons : \forall\alpha.(\alpha \rightarrowtriangle list_\alpha \rightarrowtriangle list_\alpha), nil : \forall\alpha.list_\alpha, \\ rec : \forall\alpha.((\iota_\alpha \rightarrowtriangle list_\alpha \rightarrowtriangle int) \rightarrowtriangle \iota_\alpha) \end{array} \right\}
$$

then, one can check that for any τ we have $\Gamma \vdash len : \iota_\tau \rightarrowtriangle list_\tau \rightarrowtriangle int$ and $\Gamma \vdash len(rec\;\; len) : list_\tau \rightarrowtriangle int$. Indeed, the term $len(rec\;\; len)$ takes a list whose elements are of any type, and returns an integer.

The algorithm $\mathsf{W}^{\ll}$ will compute the principal type $list_\beta \rightarrowtriangle int$ for $len(rec\,len)$.

2.4. The Algorithm $\mathsf{W}^{\ll}$

We customize the algorithm W of Damas-Milner [DM82] (see also the Caml notes of Pottier [Pot]) and we present an algorithm $\mathsf{W}^{\ll}$ that takes as input an rhoStk-term U, an environment Γ, and a set of "fresh" type variables $\mathcal{V}$, and (1) checks if it can be well-typed, and (2) infers a *principal typing* τ for U in Γ, such that :

1. The judgment $\Gamma \vdash U : \tau$ is derivable
2. If $\Gamma \vdash U : \tau'$, then there exists a substitution θ, such that $\tau' = \tau\theta$.

Definition 2.5 (The Algorithm $\mathsf{W}^{\ll}$)
The algorithm $\mathsf{W}^{\ll}$ is given in FIGURE *4 and uses the classical unification algorithm between first-order terms [JK91] (denoted* mgu *and omitted here).*

Definition 2.6 (Equality out of $\mathcal{V}$)
Two substitutions θ_1 and θ_2 are equal out of $\mathcal{V}$, written $\theta_1 \overset{\mathcal{V}}{\underset{\not\ll}{=}} \theta_2$, if $\alpha\theta_1 = \alpha\theta_2$, for all $\alpha \notin \mathcal{V}$.

Theorem 2.2 (Soundness of $\mathsf{W}^{\ll}$)
If $\mathsf{W}^{\ll}(\Gamma; U; \mathcal{V}) = (\tau; \theta; \mathcal{V}')$, then $\Gamma\theta \vdash U : \tau$.

Theorem 2.3 (Completeness and Principality of $\mathsf{W}^{\ll}$)
For all $\mathcal{V}$ and Γ, such that $\mathcal{V} \cap \mathcal{C}o\mathcal{D}om(\Gamma) = \emptyset$, if $\Gamma\phi \vdash U : \tau'$, then :

1. $\mathsf{W}^{\ll}(\Gamma; U; \mathcal{V}) \neq \mathsf{false}$ *;*

$$\boxed{\mathsf{W}^{\ll}(\Gamma; U; \mathcal{V}) = (\tau; \theta; \mathcal{V}')}$$

$$
\begin{array}{rcl}
\mathsf{W}^{\ll}(\Gamma; U; \mathcal{V}) & \triangleq & \text{match } U \text{ with} \\[4pt]
f & \Rightarrow & \text{if } f \in \mathcal{D}om(\Gamma) \text{ then} \\
 & & \text{take } (\tau; \mathcal{V}') = \mathsf{Inst}(\Gamma(f); \mathcal{V}) \text{ and } \theta = \theta_{\text{ID}} \\[4pt]
X & \Rightarrow & \text{if } X \in \mathcal{D}om(\Gamma) \text{ then} \\
 & & \text{take } (\tau; \mathcal{V}') = \mathsf{Inst}(\Gamma(X); \mathcal{V}) \text{ and } \theta = \theta_{\text{ID}} \\[4pt]
U_1 \wr U_2 & \Rightarrow & \text{let } (\tau_1; \theta_1; \mathcal{V}_1) = \mathsf{W}^{\ll}(\Gamma; U_1; \mathcal{V}) \text{ in} \\
 & & \text{let } (\tau_2; \theta_2; \mathcal{V}_2) = \mathsf{W}^{\ll}(\Gamma\theta_1; U_2; \mathcal{V}_1) \text{ in} \\
 & & \text{let } \phi = \mathsf{mgu}(\tau_1\theta_2 = \tau_2) \text{ in} \\
 & & \text{take } \tau = \tau_2\phi \text{ and } \theta = \phi \circ \theta_2 \circ \theta_1 \text{ and } \mathcal{V}' = \mathcal{V}_2 \\[4pt]
P \rightarrowtriangle U_1 & \Rightarrow & \text{let } \overline{X} = \mathcal{FV}(P) \text{ and } \overline{\alpha_X} \in \mathcal{V}_1 \text{ in} \\
 & & \text{let } (\tau_1; \theta_1; \mathcal{V}_1) = \mathsf{W}^{\ll}(\Gamma, \overline{X{:}\alpha_X}; P; \mathcal{V} \setminus \{\overline{\alpha_X}\}) \text{ in} \\
 & & \text{let } (\tau_2; \theta_2; \mathcal{V}_2) = \mathsf{W}^{\ll}(\Gamma\theta_1, \overline{X{:}\alpha_X\theta_1}; U_1; \mathcal{V}_1) \text{ in} \\
 & & \text{take } \tau = \tau_1\theta_2 \rightarrowtriangle \tau_2 \text{ and } \theta = \theta_2 \circ \theta_1 \text{ and } \mathcal{V}' = \mathcal{V}_2 \\[4pt]
U_1\, U_2 & \Rightarrow & \text{let } (\tau_1; \theta_1; \mathcal{V}_1) = \mathsf{W}^{\ll}(\Gamma; U_1; \mathcal{V}) \text{ in} \\
 & & \text{let } (\tau_2; \theta_2; \mathcal{V}_2) = \mathsf{W}^{\ll}(\Gamma\theta_1; U_2; \mathcal{V}_1) \text{ in} \\
 & & \text{let } \alpha \in \mathcal{V}_2 \text{ in} \\
 & & \text{let } \phi = \mathsf{mgu}(\tau_1\theta_2 = \tau_2 \rightarrowtriangle \alpha) \text{ in} \\
 & & \text{take } \tau = \alpha\phi \text{ and } \theta = \phi \circ \theta_2 \circ \theta_1 \text{ and } \mathcal{V}' = \mathcal{V}_2 \setminus \{\alpha\} \\[4pt]
\text{let } P \ll U_1 \text{ in } U_2 & \Rightarrow & \text{let } (\tau_1; \theta_1; \mathcal{V}_1) = \mathsf{W}^{\ll}(\Gamma; U_2; \mathcal{V}) \text{ in} \\
 & & \text{let } \overline{X} = \mathcal{FV}(P) \text{ and } \overline{\alpha_X} \in \mathcal{V}_1 \text{ in} \\
 & & \text{let } (\tau_2; \theta_2; \mathcal{V}_2) = \mathsf{W}^{\ll}(\Gamma\theta_1, \overline{X{:}\alpha_X}; P; \mathcal{V}_1 \setminus \{\overline{\alpha_X}\}) \text{ in} \\
 & & \text{let } \phi = \mathsf{mgu}(\tau_1\theta_2 = \tau_2) \text{ in} \\
 & & \text{let } (\tau_3; \theta_3; \mathcal{V}_3) = \mathsf{W}^{\ll}(\Gamma\theta_1\theta_2\phi, \overline{X{:}\mathsf{Gen}(\alpha_X\theta_2\phi; \Gamma\theta_1\theta_2\phi)}; U_1; \mathcal{V}_2) \text{ in} \\
 & & \text{take } \tau = \tau_3 \text{ and } \theta = \theta_3 \circ \phi \circ \theta_2 \circ \theta_1 \text{ and } \mathcal{V}' = \mathcal{V}_3 \\[4pt]
\_ & \Rightarrow & \text{false}
\end{array}
$$

$$\mathsf{Inst}(\forall\overline{\alpha}.\tau; \mathcal{V}) \triangleq (\tau\{\overline{\beta/\alpha}\}; \mathcal{V} \setminus \{\overline{\beta}\}) \quad \text{where } \overline{\beta} \text{ are distinct fresh variables taken in } \mathcal{V}$$

FIG. 4 – The Algorithm $\mathsf{W}^{\ll}$.

2. $\mathsf{W}^{\ll}(\Gamma; U; \mathcal{V}) = (\tau; \theta; \mathcal{V}')$, *for some* τ *and* θ *and* $\mathcal{V}'$ *;*
3. $\tau' = \tau\psi$ *and* $\phi \stackrel{\mathcal{V}}{=}_{\not{\mathcal{A}}} \psi \circ \theta$, *for some* ψ.

Theorem 2.4 (Decidability of Type Inference for rhoStk)
The following problems are decidable :

1. *Type Inference : for a closed term* U *such that* $\mathsf{stk} \notin U$, *given* Γ *(such that every constant of* U *is in* $\mathcal{D}om(\Gamma)$*), find a* τ *such that* $\Gamma \vdash U : \tau$.

2. *Type Checking : for a closed term U such that* $\mathsf{stk} \notin U$*, given* Γ *and* τ'*, check that the judgment* $\Gamma \vdash U : \tau'$ *holds.*

3. Core ML *vs.* Core rhoStk

In this section we briefly compare the syntax and semantics of the well-known core ML calculus, at the basis to the ML language, with rhoStk which could be thought as a core calculus for both ML and different rewrite-based languages. Particular focus has been put on the ratio between the theoretical tools we use *w.r.t.* the language idioms we would like to capture. The ML definitions comes from the tutorial on ML by Remy [Rém02], and the Caml notes by Pottier [Pot].

3.1. Core ML

Core ML is a fragment of ML. We recall the language and the type syntax :

Core ML

τ	$::=$	$\alpha \mid \iota \mid \tau \rightarrow \tau \mid \tau \times \tau$	Poly Types
σ	$::=$	$\forall \overline{\alpha}.\tau$	Poly Type Schemes
M	$::=$	$c \mid x \mid \lambda x.M \mid M\,M \mid \mathsf{let}\ x = M\ \mathsf{in}\ M$	Poly Terms

The operational semantics and the typing rules are the usual ones [Rém02]. Although this fragment is sufficiently expressive, and its type inference system is terminating, sound, and decidable, it lacks of useful language constructs, like recursion. To achieve this one may want to add a `fix` operator and a let rec operator. This leads to an enriched syntax :

Core ML + `fix`

τ	$::=$	As before	Poly Types
σ	$::=$	As before	Poly Type Schemes
M	$::=$	As before $\mid$ `fix`	Poly Terms + Fix

and to new static and dynamic semantics. First, we should add a new rule in the operational semantics

$$(\delta_{\mathtt{fix}}) \qquad \mathtt{fix}\,f\,v \;\rightarrow f\,(\mathtt{fix}\,f)\,v$$

where v is a term in normal form following the call-by-value strategy.

We assume the following syntactic sugar for let rec

$$\mathsf{let\ rec}\ \ f = \lambda x.M_1\,\mathsf{in}\ \ M_2 \ \ \triangleq\ \ \mathsf{let}\,f = \mathtt{fix}\ (\lambda f.\lambda x.M_1)\ \ \mathsf{in}\ \ M_2$$

and add the following typing rule for `fix`

$$\frac{\tau \equiv \tau_1 \rightarrow \tau_2}{\Gamma \vdash \mathtt{fix} : (\tau \rightarrow \tau) \rightarrow \tau}\ \text{(Fix)}$$

Even if the **Core ML** + `fix` fragment is quite expressive, it does not feature explicit matching which has been proved useful in case analysis. This extension is not so complicated : we need to enrich the type-syntax with sum-types and the calculus with appropriate injection and selection operations :

Core ML + `fix` + `match`

τ	$::=$	As before $\mid \tau + \tau$	Poly Types
σ	$::=$	As before	Poly Type Schemes
M	$::=$	As before $\mid \mathsf{inj}^n_i\ M \mid \mathtt{match}_n\, M\, M_1 \ldots M_n$	Poly Terms + Fix + Match

We should also add a new rule in the operational semantics

$$(\delta_{\mathtt{match}}) \qquad \mathtt{match}_n(\mathtt{inj}^n_i\,v)\,v_1 \ldots v_n \rightarrow v_i\,v$$

where all v's are terms in normal form following the call-by-value strategy, and add the following type rules for `inj` :

$$\frac{\Gamma \vdash M : \tau_i}{\Gamma \vdash \mathtt{inj}_i^n M : \tau_1 + \ldots + \tau_n}\ (\mathtt{inj}) \qquad \frac{\begin{array}{ll}\Gamma \vdash M : \tau_1 + \ldots + \tau_n & \\ \Gamma \vdash M_i : \tau_i \to \tau & \forall i = 1 \ldots n\end{array}}{\Gamma \vdash \mathtt{match}_n\, M\, M_1 \ldots M_n : \tau}\ (\mathtt{match})$$

This extension is morally equivalent to our rhoStk. The computability capabilities are the same, but the latter includes, as built-in features, recursion and pattern matching. This makes rhoStk a suitable candidate as a core calculus that should be more deeply compared *w.r.t.* the corresponding **Core ML** + `fix` + `match`.

4. Conclusions

In this paper we have presented a typed rewriting calculus called rhoStk which features a restricted form of polymorphism *à la* Damas-Milner-Tofte. We have customized the well-known algorithm W of Damas-Milner [DM82] for the presented calculus and we have proved the classical properties such a system should satisfy : the soundness, completeness, principality of the type inference algorithm, and thus the decidability of the type inference.

We have already shown that reduction strategies in term rewrite systems can be automatically encoded by (untyped) ρ-terms [CLW04]. The type inference algorithm we have presented here gives a correctness criterion for these terms and consequently for the encoded term rewrite system. Starting from the type system proposed in this paper we can also improve the expressiveness of the type systems usually used in rewrite-based languages with polymorphic features. This could be the case for parametric polymorphism as used in ELAN and Maude. Dealing furthermore with sub-typing is a useful open question that we are planning to study.

A useful application that still needs more investigations is to help in checking for the correctness of XML queries using the paradigms, techniques and results presented in this paper. Indeed, XML query languages like Xquery, TOM or Xcerpt (to mention just a few) are based on matching capabilities and the queries are expressed by specific rewrite rules. Of course, it will be most useful to ask the less contextual informations to the user nd to use the DTD or the XML schema together with type inference to help the user in making safer and meaningful queries [CCD+05]. Such investigations are underway in languages like CDUCE [BCF03, HFC05] and we believe that the polymorphic rewriting calculus can bring a useful framework towards the useful interactions of matching and polymorphic types for XML.

Références

[Bar84] H. Barendregt. *Lambda Calculus : its Syntax and Semantics*. North Holland, 1984.

[BCF03] V. Benzaken, G. Castagna, and A. Frisch. Cduce : an xml-centric general-purpose language. In C. Runciman and O. Shivers, editors, *ICFP*, pages 51–63. ACM, 2003.

[BCKL03] G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure Patterns Type Systems. In *Proc. of POPL*, pages 250–261. The ACM press, 2003.

[BCM04] D. A. Basin, M. Clavel, and J. Meseguer. Reflective metalogical frameworks. *ACM Trans. Comput. Log.*, 5(3) :528–576, 2004.

[Cas96] C. Castro. Solving Binary CSP using Computational Systems. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. ENTCS.

[CCD+05] H. Cirstea, E. Coquery, W. Drabent, F. Fages, C. Kirchner, L. Liquori, B. Wack, and A. Wilk. Types for Rewerse Reasoning and Query Languages. Deliverable, 2005. REWERSE Network of Excellence.

[CKL01a] H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In *Proc. of RTA*, volume 2051 of *LNCS*, pages 77–92. Springer-Verlag, 2001.

[CKL01b] H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In *Proc. of FOSSACS*, volume 2030 of *LNCS*, pages 166–180, 2001.

[CLW04] H. Cirstea, L. Liquori, and B. Wack. Rewriting calculus with fixpoints : Untyped and first-order systems. In *TYPES'03*, volume 3085 of *Lecture Notes in Computer Science*, Torino, 2004.

[CKLW06] H. Cirstea, C. Kirchner, L. Liquori, and B. Wack. Decidable Type Inference for the Polymorphic Rewriting Calculus. Technical Report inria-00000817, Loria, 2006. Available at `http://hal.inria.fr`

[CMR04] H. Cirstea, P.-E. Moreau, and A. Reilles. Rule based programming in java for protocol verification. In *Workshop on Rewriting Logic and Applications*, Barcelona (Spain), March 2004. Electronic Notes in Theoretical Computer Science.

[Cur34] H. Curry. Functionality in Combinatory Logic. In *Proc. Nat. Acad. Sci. U.S.A.*, volume 20, pages 584–590, 1934.

[DM82] L. Damas and R. Milner. Principal Type-Schemes for Functional Programs. In *Proc. of POPL*, pages 207–212. The ACM Press, 1982.

[EMS03] S. Eker, J. Meseguer, and A. Sridharanarayanan. The maude ltl model checker and its implementation. In T. Ball and S. K. Rajamani, editors, *SPIN*, Lecture Notes in Computer Science, pages 230–234. Springer, 2003.

[FN96] K. Futatsugi and A. Nakagawa. An Overview of Cafe Project. In *Proc. of CafeOBJ Workshop*, 1996.

[Gog04] J. Goguen. The OBJ Family Home Page, 2005. `http://www.cs.ucsd.edu/users/goguen/sys/obj.html`.

[GR88] P. Giannini and S. Ronchi della Rocca. Characterization of Typings in Polymorphic Type Discipline. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, pages 61–70, 1988.

[HFC05] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for xml. In J. Palsberg and M. Abadi, editors, *POPL*, pages 50–62. ACM, 2005.

[JK91] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras : a rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991.

[KMR05] C. Kirchner, P.-E. Moreau, and A. Reilles. Formal validation of pattern matching code. In *PPDP '05 : Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 187–197, New York, NY, USA, 2005. ACM Press.

[Lei83] D. Leivant. Polymorphic Type Inference. In *Proc. of POPL*, pages 88–98. The ACM Press, 1983.

[LW05] L. Liquori and B. Wack. The polymorphic rewriting-calculus : Type checking vs. type inference. In N. Martí-Oliet, editor, *Proceedings of the Fifth International Workshop on Rewriting Logic and Its Application*, volume 117 of *entcs*, pages 89 – 111. Elsevier B. V., 2005.

[MOM02] N. Martí-Oliet and J. Meseguer. Rewriting Logic : Roadmap and Bibliography. *Theoretical Computer Science*, 285(2) :121–154, 2002.

[MRV03] P.-E. Moreau, C. Ringeissen, and M. Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, May 2003.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[Pey87] S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[Pot] F. Pottier. Course DEA : Typage et programmation. `http://pauillac.inria.fr/~fpottier/mpri/dea-typage.ps.gz`.

[Rém02] D. Rémy. Using, Understanding, and Unraveling the OCaml Language. In *Applied Semantics. Advanced Lectures. LNCS 2395.*, pages 413–537. Springer Verlag, 2002.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *journal of the ACM*, 12 :23–41, 1965.

[Ste02] M.-O. Stehr. *Programming, Specification and Interactive Theorem Proving – Towards a Unified Language based on Equational Logic, Rewriting Logic and Type Theory.* PhD thesis, Universität Hamburg, Fachbereich Informatik, 2002.

[The03a] The Cristal Team. The Objective Caml Home Page, 2003. `http://www.ocaml.org/`.

[The03b] The GNU Prolog Team. The GNU Prolog Home Page, 2003. `http://pauillac.inria.fr/~diaz/gnu-prolog/`.

[The04a] The Asf+Sdf Team. The Asf+Sdf Meta-Environment Home Page, 2005. `http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/MetaEnvironment`.

[The04b] The Haskell Team. The Haskell Home Page, 2005. `http://www.haskell.org/`.

[The04c] The Maude Team. The Maude Home Page, 2005. `http://maude.cs.uiuc.edu/`.

[The04d] The Protheo Team. The Elan Home Page, 2005. `http://elan.loria.fr`.

[The04e] The Scheme Team. The Scheme Language, 2005. `http://www.swiss.ai.mit.edu/projects/scheme/`.

[vdBKV96] M. van den Brand, P. Klint, and C. Verhoef. Core technologies for system renovation. In K. G. Jeffery, J. Král, and M. Bartosek, editors, *SOFSEM : Theory and Practice of Informatics, 23rd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1175 of *Lecture Notes in Computer Science*, pages 235–254. Springer, 1996.

[vO90] V. van Oostrom. Lambda Calculus with Patterns. Technical Report IR-228, Faculteit der Wiskunde en Informatica, Vrije Universiteit Amsterdam, 1990.

[Wac05] B. Wack. *Typage et déduction dans le calcul de réécriture*. Thèse de doctorat, Université Henri Poincaré - Nancy I, Oct 2005.

[Wel99] J. B. Wells. Typability and Type Checking in System F are Equivalent and Undecidable. *Annals of Pure and Applied Logic*, 98(1–3) :111–156, 1999.