Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 117 (2005) 89–111
www.elsevier.com/locate/entcs

The Polymorphic Rewriting-calculus
[Type Checking *vs.* Type Inference]

Luigi Liquori[a] and Benjamin Wack[b]

[a] *INRIA Sophia Antipolis*

[b] *LORIA & Université Henri Poincaré, Nancy*

Abstract

The Rewriting-calculus (Rho-calculus), is a minimal framework embedding Lambda-calculus and Term Rewriting Systems, by allowing abstraction on variables and patterns. The Rho-calculus features higher-order functions (from Lambda-calculus) and pattern-matching (from Term Rewriting Systems). In this paper, we study extensively a second-order Rho-calculus *à la* Church (RhoF) that enjoys subject reduction, type uniqueness, and decidability of typing. Then we apply a classical type-erasing function to RhoF, to obtain an untyped Rho-calculus *à la* Curry (uRhoF). The related type inference system is isomorphic to RhoF and enjoys subject reduction. Both RhoF and uRhoF systems can be considered as minimal calculi for polymorphic rewriting-based programming languages. We discuss the possibility of a logic existing underneath the type systems via a Curry-Howard Isomorphism.

Keywords: Rewriting-calculus, Pattern-matching, Polymorphism, Type-checking *vs.* Type-inference.

1 Introduction

A promising line of research unifying the logic paradigm with the functional paradigm is that of *rewriting-based* languages [29] (Elan[39], Maude [38], ASF+SDF[41,36], OBJ*[17,20], ...). Although these languages are less used than object-oriented languages (Java[33], C#[27], ...), they can also serve as (formal) common intermediate languages for implementing compilers for rewriting-based, functional, object-oriented, logic, and other "high-level" modern languages.

One of the main advantages of the rewriting-based languages is *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. Another advantage of rewriting-based languages (in contrast with ML or Haskell) is the ability to handle non-determinism by means of a collection of results: pattern-matching needs not to be exclusive, *i.e.* multiple branches can be taken simultaneously. An empty collection of results represents a matching failure, a singleton represents a deterministic result, and a collection with more than one element represents a non-deterministic choice between the elements of the collection.

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[30,34], Haskell[37], Scheme[40], or Prolog[35]. 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 an *ad hoc* paradigm of computation; we argue that the computational behavior of a calculus can be really influenced by the presence of pattern-matching.

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 [31,42], 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 [32].

The *Rewriting-calculus* [9,12] integrates matching, rewriting 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 pattern P. Note that the Rewriting-calculus is a generalization of the Lambda-calculus, since we get the Lambda-calculus back if every pattern P is a variable.

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. One of the advantages of matching is that it can be customized with elaborated equational theories.

A large class of Term Rewriting Systems are also embedded in the Rewriting-calculus. In particular, the notions of *rule application* and *result* (basic ingredients of Term Rewriting Systems) become explicit. A rewrite rule is a first-class citizen, which can be created, manipulated and modified by the calculus itself. The abilities to manipulate rules and to define evaluation strategies represent the basic methods in rewriting-based languages. These strategies can be *implicit* as in ASF+SDF[41], *local* as in OBJ* and Maude, or *user defined* as in Elan and Maude[13]. Previous papers [7,8,12] showed that the Rewriting-calculus can be used as a core engine calculus for rewriting-based languages such as Elan and Maude.

Finally, *static analysis* via a type system (inherited from the Lambda-calculus) enforces a safer programming discipline. This paper presents a Rho-calculus *à la* Church (called RhoF) featuring second-order polymorphic types. Then we apply a classical erasing function to RhoF in order to obtain a corresponding type inference system *à la* Curry (uRhoF). We discuss the possibility of a logic existing underneath both type systems via the well-known Curry-Howard Isomorphism. The two systems are good candidates for giving the static semantics of a family of rewriting-based languages such as Elan and Maude.

The main contributions of this paper are:

- to present and study a fully-typed Polymorphic Rewriting-calculus (RhoF); this type system could be used to give a static type discipline for fully typed rewriting-based languages.
- to study the corresponding Type Inference Polymorphic Rewriting-calculus (uRhoF) obtained from RhoF via a type-erasing function;
- to discuss a possible common logic underneath the two type systems.

Plan of the paper.

In Section 2, we present the syntax of the calculus and its small-step semantics. In Section 3, we introduce the fully typed second-order rewriting calculus *à la* Church RhoF: types of the bound variables are specified in the term, making type reconstruction and verification quite straightforward. The calculus enjoys subject reduction, and type uniqueness. In Section 4, we present the calculus *à la* Curry uRhoF: type information is not given in the term, and the type system is not fully syntax-directed, thus enforcing a flexible polymorphic type discipline. The calculus enjoys subject reduction, but as it is well-known for the λ-calculus, type inference is undecidable. Section 5 conclude.

2 The System RhoF

We detail the syntax and the semantics of RhoF, and we give some examples.

2.1 Syntax

Notational Conventions.

We consider the meta-symbols "$\_ \twoheadrightarrow \_$" (function- and type-abstraction), and "$[\_ \ll \_]$" (delayed matching constraint), and "$\_ \,;\, \_$" (structure operator).

Syntactic Cat.	**Abstract Syntax**		
$K \in \mathcal{K}inds$	K	$::=$	$*$
$\tau.\iota \in \mathcal{T}ype$	τ	$::=$	$\iota \mid \alpha \mid \tau \rightharpoonup \tau \mid \forall\alpha.\tau$
$\Gamma.\Delta \in \mathcal{C}ontext$	Δ	$::=$	$\emptyset \mid \Delta, \alpha{:}K \mid \Delta, f{:}\tau \mid \Delta, X{:}\tau$
$P.Q \in \mathcal{P}attern$	P	$::=$	$\mathsf{stk} \mid \alpha \mid X \mid f(\overline{P})$ *(all vars occur only once in any P)*
$A.B.f \in \mathcal{T}erm$	A	$::=$	$\mathsf{stk} \mid f \mid X \mid P \rightharpoonup_\Delta A \mid [P \ll_\Delta A]A \mid A\ A \mid A\ \tau \mid A\,;A$

Figure 1. Syntax of RhoF

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 "$[\ll]$" which is higher than that of "$\rightharpoonup$ " which is, in turn, of higher priority than the ";". 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 ($\mathcal{T}ype_K \subseteq \mathcal{T}ype$), the symbols α, β range over the set $\mathcal{T}ype_V$ of type-variables ($\mathcal{T}ype_V \subseteq \mathcal{T}ype$), the symbols $A, B, C, \ldots, U, V, W, Z$ range over the set $\mathcal{T}erm$ of (un)typed terms, the symbols $X, Y, Z, \ldots$ range over the set $\mathcal{V}ar$ of term variables ($\mathcal{V}ar \subseteq \mathcal{T}erm$), the symbols $a, b, c, \ldots, f, g, h, \ldots$ range over a set $\mathcal{T}erm_K$ of term constants ($\mathcal{T}erm_K \subseteq \mathcal{T}erm$). The symbols P, Q range over the set $\mathcal{P}attern$ of patterns, ($\mathcal{V}ar \subseteq \mathcal{P}attern \subseteq \mathcal{T}erm$). The symbols θ, ϕ, ψ range over substitutions. Finally, the symbols $\mathcal{A}, \mathcal{B}, \mathcal{C}$ range over $\mathcal{T}ype \cup \mathcal{T}erm$. 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 currified in order to denote a function taking multiple arguments, *e.g.* $f(\overline{A}) \triangleq f(A_1, \cdots, A_n) \triangleq f\ A_1 \cdots A_n$.

Syntax (Figure 1).

The *types* are as one would expect from 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 rewriting rules; the set of patterns is obviously included in the set of terms. The well-known linearity restriction [42] is needed to keep the small-step semantics confluent. A typed *rewriting rule* of the form $P \rightharpoonup_\Delta A$ abstracting over the free-variables of P is a first-class citizen of the calculus. The context Δ records the type of the free-variables of P (bound in A). When a pattern is a variable, we write $X \twoheadrightarrow_\tau A$, instead of $X \twoheadrightarrow_{(X:\tau)} A$ (by a little abuse of notation). An *application* is implicitly denoted by concatenation; note that *"terms can be applied to types"*. The *delayed matching constraint* $[P \ll_\Delta A]B$ can be seen as the term B with its free-variables (declared in Δ) constrained by the matching between P and A. The symbol stk is the special constant representing all the delayed matching constraints whose matching problem is unsolvable. A *structure* is a collection of terms that can be seen either as a set of rewriting rules or as a set of results.

2.1.1 Free-Variables and Substitutions.

Definition 2.1 [Free-variables Fv]

$$\begin{array}{lcllcl}
\mathsf{Fv}(f) & \triangleq & \emptyset & \mathsf{Fv}(P \twoheadrightarrow_\Delta A) & \triangleq & \mathsf{Fv}(A) \cup \mathsf{Fv}(\Delta) \setminus \mathsf{Fv}(P) \\
\mathsf{Fv}(\mathsf{stk}) & \triangleq & \emptyset & \mathsf{Fv}([P \ll_\Delta \mathcal{A}]B) & \triangleq & \mathsf{Fv}((P \twoheadrightarrow_\Delta B)\ \mathcal{A}) \\
\mathsf{Fv}(X) & \triangleq & \{X\} & \mathsf{Fv}(A\,;B/A\ B) & \triangleq & \mathsf{Fv}(A) \cup \mathsf{Fv}(B) \\
\mathsf{Fv}(\alpha) & \triangleq & \{\alpha\} & \mathsf{Fv}(A\ \tau) & \triangleq & \mathsf{Fv}(A) \cup \mathsf{Fv}(\tau) \\
 & & & \mathsf{Fv}(\tau_1 \twoheadrightarrow \tau_2) & \triangleq & \mathsf{Fv}(\tau_1) \cup \mathsf{Fv}(\tau_2)
\end{array}$$

As usual, we work modulo α*-conversion* and we adopt Barendregt's *"hygiene-convention"* [1], *i.e.* free- and bound-variables have different names. This allows us to define substitutions quite straightforwardly, since it avoids problems like variable capture.

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 $\{A_1/X_1 \ldots A_m/X_m\}$, or $\{\tau_1/\alpha_1 \ldots \tau_m/\alpha_m\}$, and its domain $\mathsf{Dom}(\theta)$ denotes $\{X_1, \ldots, X_m\}$, resp. $\{\alpha_1, \ldots, \alpha_m\}$. The application of a substitution θ to a term A (resp. type τ), denoted by $A\theta$ (resp. $\tau\theta$), is defined as follows:

$$\begin{array}{lcllcl}
f\theta & \triangleq & f & (P \twoheadrightarrow_\Delta A)\theta & \triangleq & P \twoheadrightarrow_\Delta A\theta \\
\mathsf{stk}\theta & \triangleq & \mathsf{stk} & ([P \ll_\Delta \mathcal{A}]B)\theta & \triangleq & [P \ll_\Delta \mathcal{A}\theta]B\theta \\
(A\ \tau)\theta & \triangleq & (A\theta)\ (\tau\theta) & (A\,;B/A\ B)\theta & \triangleq & A\theta\,;B\theta/A\theta\ B\theta \\
X_i\theta & \triangleq & \begin{cases} A_i & \text{if} X_i \in \mathsf{Dom}(\theta) \\ X_i & \text{otherwise} \end{cases} & \alpha_i\theta & \triangleq & \begin{cases} \tau_i & \text{if } \alpha_i \in \mathsf{Dom}(\theta) \\ \alpha_i & \text{otherwise} \end{cases} \\
\iota\theta & \triangleq & \iota & (\tau_1 \twoheadrightarrow \tau_2)\theta & \triangleq & \tau_1\theta \twoheadrightarrow \tau_2\theta
\end{array}$$

2.1.2 Matching Equations, Theories and Term Approximations.

The core mechanism of the Rewriting-calculus is pattern-matching. When a delayed matching constraint is evaluated, then a corresponding matching problem has to be solved. We use a theory for the Rho-calculus (introduced in [12]) that handles uniformly matching failures and eliminates them when not significant for the computation. We define rules for handling this kind of terms and we show how they are integrated in the calculus. The classical notions of matching equations and matching solutions are defined as usual.

Definition 2.3 [Matching]
Given a theory $\mathbb{T}$

(i) A *matching equation* is a problem $\mathsf{T} \triangleq P \prec\!\!\prec_{\mathbb{T}} A$ where P is a pattern and A is a term;

(ii) A substitution θ is a solution of the matching equation T if $P\theta \overset{\mathbb{T}}{=} A$.

Different theories and the corresponding pattern-matching problems can be formally defined and solved, for example as explained in [8]. If the equation $P \prec\!\!\prec_{\mathbb{T}} A$ has a unique solution, we denote it by $\theta_{(P \prec\!\!\prec_{\mathbb{T}} A)}$.

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

Definition 2.4 [Superposition]

(i) The relation of superposition $P \sqsubseteq A$ is defined according to the structure of P as follows:

$$\begin{array}{lcll} f & \sqsubseteq & f & \\ \mathsf{stk} & \sqsubseteq & \mathsf{stk} & \\ X & \sqsubseteq & A & (\forall A) \\ \alpha & \sqsubseteq & \tau & (\forall \tau) \end{array} \qquad \begin{array}{l} f(\overline{P}) \;\sqsubseteq\; A \;\text{ if } A \equiv f(\overline{B}) \wedge \overline{P} \sqsubseteq \overline{B} \\ P \;\sqsubseteq\; A \;\text{ if } A \equiv \begin{cases} X \vee (A_1\,; A_2) \vee A\ \tau\ \vee \\ (A_1\ A_2 \wedge A_1 \notin \mathcal{P}attern) \vee \\ ([Q \ll_{\Delta} A_1]A_2 \wedge Q \sqsubseteq A_1 \wedge P \sqsubseteq A_2)\ (\forall P) \end{cases} \end{array}$$

(ii) If $P \sqsubseteq A$ is not satisfied we write $P \not\sqsubseteq A$.

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

Definition 2.5 [Stuck Theory, $\mathbb{T}_{\mathsf{stk}}$]

$$(P \rightarrowtriangle_{\Delta} A)\ \mathcal{B} \ \to_{\rho}\ [P \ll_{\Delta} \mathcal{B}]A$$

$$[P \ll_{\Delta} \mathcal{B}]A \ \to_{\sigma}\ A\theta_{(P \prec\!\!\prec_{\mathsf{stk}} \mathcal{B})}$$

$$(A\,;B)\ C \ \to_{\delta}\ A\ C\,;B\ C$$

$$A \to_{\mathsf{stk}} B$$

Figure 2. Top-level Rules of RhoF

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

$$\begin{array}{lll}
[P \ll_{\Delta} A]B & \to_{\mathsf{stk}}\ \mathsf{stk} & \text{if } P \not\sqsubseteq A \\
\mathsf{stk}\,; A & \to_{\mathsf{stk}}\ A & \\
A\,; \mathsf{stk} & \to_{\mathsf{stk}}\ A & \\
\mathsf{stk}\ A & \to_{\mathsf{stk}}\ \mathsf{stk} &
\end{array}$$

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

As mentioned previously, these rules are used to propagate or eliminate the definitively stuck terms.

2.2 The Polymorphic Rewriting-calculus, RhoF

Figure 2 shows the reduction rules of RhoF parameterized by the theory $\mathbb{T}_{\mathsf{stk}}$ (recall the symbols $\mathcal{A}, \mathcal{B}, \mathcal{C}$ range over $\mathcal{T}ype \cup \mathcal{T}erm$).

Let us quickly explain the top-level rules:

(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 is applied if and only if the matching equation $P \prec\!\!\prec_{\mathsf{stk}} \mathcal{B}$ has at least one solution: in this case the matching solutions are computed and applied to the term A. If there is more than one match, a structure collecting all the different results is obtained when the rule is applied. If there is no solution, this rule does not apply and thus, the term that is on the left-hand side represents a matching failure. As we shall see, further reductions or instantiations are likely to modify $\mathcal{B}$ 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 rewriting rules that are particular to the theory adopted in the calculus; in our case the above defined $\mathbb{T}_{\mathsf{stk}}$-theory.

We denote by $\mapsto_{\rho\sigma\delta}$ the contextual closure induced by these rules. Its reflexive and transitive closure is denoted by $\mapsto\!\!\!\!\rightarrow_{\rho\sigma\delta}$. The symmetric and transitive closure of $\mapsto\!\!\!\!\rightarrow_{\rho\sigma\delta}$ is denoted by $=_{\rho\sigma\delta}$. Notice that these relations are parameterized by the adopted theory $\mathbb{T}_{\mathsf{stk}}$. We denote by $\mapsto^{\mathsf{stk}}_{\rho\sigma\delta}$ the relation $\mapsto_{\mathsf{stk}} \cup \mapsto_{\rho\sigma\delta}$. For $\mapsto^{\mathsf{stk}}_{\rho\sigma\delta}$, the following holds.

Theorem 2.6 (Church Rosser for RhoF [12])
The relation $\mapsto^{\mathsf{stk}}_{\rho\sigma\delta}$ is confluent.

3 The Polymorphic Type System RhoF

Types can be used as predicates for terms of Rho-calculus. Terms can be directly *decorated* with types and then every closed term comes directly with a unique, intrinsic type. In this *fully typed* approach, a type judgment will be denoted by the symbol $\vdash_{\mathsf{T}}$ (for Typed terms). A *typed system* is a set of rules for proving judgments of the shape $\Gamma \vdash_{\mathsf{T}} A : \tau$, where A is a typed term, τ is a type, and Γ is a context. The meaning of such a judgment is: the term A has type τ under the context Γ, and Γ records the types of the free-variables of Γ and τ. Figures 3, and 4 presents the kinding/typing rules of RhoF, which are directly inspired by the Girard System F [19]. More precisely, the system proves judgment of the shape:

$$\Gamma \vdash_{\mathsf{T}} ok \quad \text{and} \quad \Gamma \vdash_{\mathsf{T}} \tau : * \quad \text{and} \quad \Gamma \vdash_{\mathsf{T}} P : \tau \quad \text{and} \quad \Gamma \vdash_{\mathsf{T}} A : \tau$$

We discuss only the typing rules for well-formed terms and patterns, the other typing rules being standard.

- $(Term{\cdot}Var)(Term{\cdot}Const)$**:** As usual, the context determines the type of variables. It cannot contain two declarations for the same variable (or constant);
- $(Term{\cdot}Stuck)$**:** Since stk can appear in any structure, its type can be virtually anything but *falsum*, *i.e.* $\bot \triangleq \forall\alpha.\alpha$;
- $(Term{\cdot}Abs^{\rightarrow})$**:** For the left-hand side of the arrow-type, we use the type of the pattern P; this rule allows one to hide some type information in a pattern containing applications (*e.g.* τ_2 disappears in the final type of $f(X)$ in the judgment $f{:}\tau_2 \rightarrow \tau_1, X{:}\tau_2 \vdash f(X) : \tau_1$). The context Δ gives the types of the free-variables of P. The type system ensures that the solutions of the corresponding matching equations are well-typed;

Well-formed Contexts

$$\frac{}{\emptyset \vdash_{\mathsf{T}} ok}\,(Ctx{\cdot}Empty) \qquad \frac{\Gamma \vdash_{\mathsf{T}} ok \quad \alpha \notin \mathsf{Dom}(\Gamma)}{\Gamma, \alpha{:}{*} \vdash_{\mathsf{T}} ok}\,(Ctx{\cdot}Var^{\forall})$$

$$\frac{\Gamma \vdash_{\mathsf{T}} ok \quad \iota \notin \mathsf{Dom}(\Gamma)}{\Gamma, \iota{:}{*} \vdash_{\mathsf{T}} ok}\,(Ctx{\cdot}Const) \qquad \frac{\Gamma \vdash_{\mathsf{T}} ok \quad \Gamma \vdash_{\mathsf{T}} \tau : * \quad X \notin \mathsf{Dom}(\Gamma)}{\Gamma, X{:}\tau \vdash_{\mathsf{T}} ok}\,(Ctx{\cdot}Var)$$

Well-kinded Types

$$\frac{\Gamma_1, \iota{:}{*}, \Gamma_2 \vdash_{\mathsf{T}} ok}{\Gamma_1, \iota{:}{*}, \Gamma_2 \vdash_{\mathsf{T}} \iota : *}\,(Type{\cdot}Const) \qquad \frac{\Gamma_1, \alpha{:}{*}, \Gamma_2 \vdash_{\mathsf{T}} ok}{\Gamma_1, \alpha{:}{*}, \Gamma_2 \vdash_{\mathsf{T}} \alpha : *}\,(Type{\cdot}Var)$$

$$\frac{\Gamma, \alpha{:}{*} \vdash_{\mathsf{T}} \tau : *}{\Gamma \vdash_{\mathsf{T}} \forall \alpha.\tau : *}\,(Type{\cdot}Poly) \qquad \frac{\Gamma \vdash_{\mathsf{T}} \tau_1 : * \quad \Gamma \vdash_{\mathsf{T}} \tau_2 : *}{\Gamma \vdash_{\mathsf{T}} \tau_1 \twoheadrightarrow \tau_2 : *}\,(Type{\cdot}Arrow)$$

Figure 3. The Kind System for RhoF

- $(Term{\cdot}Appl)$**:** We directly exploit the information given in the type of the function, statically checking that the given argument has the expected type τ_1;
- $(Term{\cdot}Abs^{\forall})$**:** The rationale is: $\alpha \twoheadrightarrow_{*} A \simeq \alpha \twoheadrightarrow_{(\alpha:*)} A$. Abstraction on type-variables makes the polymorphic mechanism available at the user-level: note that a trivial pattern is used in polymorphic-abstraction.
- $(Term{\cdot}Appl^{\forall})$**:** The rationale is: all free occurrences of α in τ_1 are substituted with τ_2. Any well-formed type τ_2 is suitable, which makes the typing fully polymorphic.
- $(Term{\cdot}Struct)$**:** This rule states that all the members of a structure have the same type. This is important when considering structures as a collection of results; if a function can return different results, then we would at least expect them to have the same type;
- $(Term{\cdot}Match^{\twoheadrightarrow})(Term{\cdot}Match^{\forall})$**:** The first rule states that the constraint $[P \ll_{\Delta} \mathcal{B}]A$ gets the same type as $(P \twoheadrightarrow_{\Delta} A)\ \mathcal{B}$. This is sound since $(P \twoheadrightarrow_{\Delta} A)\ \mathcal{B} \rightarrow_{\rho} [P \ll_{\Delta} \mathcal{B}]A$. The second rule instantiates α with τ_2.

Example 3.1 [Some derivable typing judgments [2]-inspired]

Well-formed Terms and Patterns

$$\frac{\Gamma_1, X{:}\tau, \Gamma_2 \vdash_{\mathsf{T}} ok}{\Gamma_1, X{:}\tau, \Gamma_2 \vdash_{\mathsf{T}} X : \tau}\,(Term{\cdot}Var) \qquad \frac{\Gamma_1, f{:}\tau, \Gamma_2 \vdash_{\mathsf{T}} ok}{\Gamma_1, f{:}\tau, \Gamma_2 \vdash_{\mathsf{T}} f : \tau}\,(Term{\cdot}Const)$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{T}} A : \tau_1 \twoheadrightarrow \tau_2\\ \Gamma \vdash_{\mathsf{T}} B : \tau_1\end{array}}{\Gamma \vdash_{\mathsf{T}} A\ B : \tau_2}\,(Term{\cdot}Appl^{\twoheadrightarrow}) \qquad \frac{\begin{array}{l}\Gamma, \Delta \vdash_{\mathsf{T}} P : \tau_1 \quad \mathsf{Fv}(P) = \mathsf{Dom}(\Delta)\\ \Gamma, \Delta \vdash_{\mathsf{T}} A : \tau_2 \quad \Gamma, \Delta \vdash_{\mathsf{T}} \tau_1 \twoheadrightarrow \tau_2 : *\end{array}}{\Gamma \vdash_{\mathsf{T}} P \twoheadrightarrow_{\Delta} A : \tau_1 \twoheadrightarrow \tau_2}\,(Term{\cdot}Abs^{\twoheadrightarrow})$$

$$\frac{\Gamma, \alpha{:}* \vdash_{\mathsf{T}} A : \tau}{\Gamma \vdash_{\mathsf{T}} \alpha \twoheadrightarrow_{*} A : \forall\alpha.\tau}\,(Term{\cdot}Abs^{\forall}) \qquad \frac{\Gamma \vdash_{\mathsf{T}} A : \forall\alpha.\tau_1 \quad \Gamma \vdash_{\mathsf{T}} \tau_2 : *}{\Gamma \vdash_{\mathsf{T}} A\ \tau_2 : \tau_1\{\tau_2/\alpha\}}\,(Term{\cdot}Appl^{\forall})$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{T}} A : \tau\\ \Gamma \vdash_{\mathsf{T}} B : \tau\end{array}}{\Gamma \vdash_{\mathsf{T}} A ; B : \tau}\,(Term{\cdot}Struct) \qquad \frac{\begin{array}{l}\Gamma, \Delta \vdash_{\mathsf{T}} P : \tau_1 \quad \mathsf{Fv}(P) = \mathsf{Dom}(\Delta)\\ \Gamma, \Delta \vdash_{\mathsf{T}} A : \tau_2 \quad \Gamma \vdash_{\mathsf{T}} B : \tau_1\end{array}}{\Gamma \vdash_{\mathsf{T}} [P \ll_{\Delta} B]A : \tau_2}\,(Term{\cdot}Match^{\twoheadrightarrow})$$

$$\frac{\Gamma \vdash_{\mathsf{T}} \tau : * \quad \tau \neq \bot}{\Gamma \vdash_{\mathsf{T}} \mathsf{stk} : \tau}\,(Term{\cdot}Stuck) \qquad \frac{\Gamma \vdash_{\mathsf{T}} \tau_2 : * \quad \Gamma, \alpha{:}* \vdash_{\mathsf{T}} A : \tau_1}{\Gamma \vdash_{\mathsf{T}} [\alpha \ll_{*} \tau_2]A : \tau_1\{\tau_2/\alpha\}}\,(Term{\cdot}Match^{\forall})$$

Figure 4. The Type System for RhoF

Let $\Gamma \triangleq \iota{:}*, f{:}\iota \twoheadrightarrow \iota, a{:}\iota$. The following judgments are derivable:

$\emptyset \vdash_{\mathsf{T}} \bot \equiv \forall\alpha.\alpha : *$	Second-order definition of *falsum*
$\emptyset \vdash_{\mathsf{T}} \alpha \twoheadrightarrow_{*} X \twoheadrightarrow_{\bot} (X\ \alpha) : \forall\alpha.(\bot \twoheadrightarrow \alpha)$	*Ex falso sequitur quodlibet*[1]
$\emptyset \vdash_{\mathsf{T}} \beta \twoheadrightarrow_{*} Y \twoheadrightarrow_{\beta} X : \forall\beta.(\beta \twoheadrightarrow \beta)$	Polymorphic identity
$\Gamma \vdash_{\mathsf{T}} (\gamma \twoheadrightarrow_{*} f(Z) \twoheadrightarrow_{(Z:\gamma)} Z)\ \iota\ f(a) : \iota$	Polymorphic instantiation-application

3.1 Metatheory of RhoF

The type system ensures that arguments of a function have the same types as the corresponding formal parameters. The rule ($Term{\cdot}Appl$) only checks that the pattern expected by a function and the argument (considered as a single term) have the same type. The shape of pattern is essential to guarantee the

[1] Anything follows from a false judgment: the subject of this judgment is its proof.

soundness of the type system: the more expressive the patterns are, the more non-sense can follow.

Remark 3.2 [Spoofer [3]] If we allow variables as the head symbol of a pattern (called "active variables"), then we can write the following counter-example. In the context $\Gamma \triangleq X{:}\tau_1 \twoheadrightarrow \tau_2, Y{:}\tau_1, f{:}\tau_3 \twoheadrightarrow \tau_2, a{:}\tau_3$, the pattern

$$\frac{\Gamma \vdash_{\mathsf{T}} X : \tau_1 \twoheadrightarrow \tau_2 \quad \Gamma \vdash_{\mathsf{T}} Y : \tau_1}{\Gamma \vdash_{\mathsf{T}} X(Y) : \tau_2} \quad \text{and the term} \quad \frac{\Gamma \vdash_{\mathsf{T}} f : \tau_3 \twoheadrightarrow \tau_2 \quad \Gamma \vdash_{\mathsf{T}} a : \tau_3}{\Gamma \vdash_{\mathsf{T}} f(a) : \tau_2}$$

have a common type, but the solution of the matching problem $X(Y) \lll f(a)$ instantiates X and Y with terms not having the expected type (*i.e.* subject reduction is lost); if $\Gamma \triangleq f{:}\tau_3 \twoheadrightarrow \tau_2, a{:}\tau_3$ and $\Delta \triangleq X{:}\tau_1 \twoheadrightarrow \tau_2, Y{:}\tau_1$, then $\Gamma \vdash_{\mathsf{T}} (X(Y) \twoheadrightarrow_{\Delta} Y) f(a) : \tau_1$ but $\Gamma \vdash_{\mathsf{T}} a : \tau_3$.

All the metaproperties presented below for RhoF are adapted from the classical properties of the Girard's Lambda-calculus.

Lemma 3.3 (Substitution Lemma)

(i) *If* $\Gamma, \Delta \vdash_{\mathsf{T}} P : \tau$ *and* $\Gamma \vdash_{\mathsf{T}} B : \tau$ *and* $\mathsf{Dom}(\Delta) = \mathsf{Fv}(P)$, *are such that* $P \lll_{\mathsf{stk}} B$ *has a solution* θ, *then for all* $X \in \mathsf{Fv}(P)$, *there exists* σ *such that* $\Gamma, \Delta \vdash_{\mathsf{T}} X : \sigma$ *and* $\Gamma \vdash_{\mathsf{T}} X\theta : \sigma$.

(ii) *If* $\Gamma, \Delta \vdash_{\mathsf{T}} A : \tau$, *then for any well-typed substitution* θ *such that* $\mathsf{Dom}(\theta) = \mathsf{Dom}(\Delta)$, *we have* $\Gamma \vdash_{\mathsf{T}} A\theta : \tau$.

Theorem 3.4 (Subject Reduction for RhoF)
If $\Gamma \vdash_{\mathsf{T}} A : \tau$ *and* $A \mapsto^{\mathsf{stk}}_{\rho\sigma\delta} B$, *then* $\Gamma \vdash_{\mathsf{T}} B : \tau$.

Proof *By an induction on the derivation of* $\Gamma \vdash_{\mathsf{T}} A : \tau$. □

Theorem 3.5 (Type Uniqueness for $\rho^{\mathsf{Fix}}_{\twoheadrightarrow}$**)**
If $\Gamma \vdash_{\mathsf{T}} A : \tau_1$ *and* $\Gamma \vdash_{\mathsf{T}} A : \tau_2$, *and* $\mathsf{stk} \notin A$, *then* $\tau_1 \equiv \tau_2$.

Proof *By an easy induction on the structure of A.* □

Theorem 3.6 (Decidability of Typing for RhoF)
For a closed A *such that* $\mathsf{stk} \notin A$, *the following problems are decidable:*

(i) *Type Reconstruction: is there a type* τ *such that* $\emptyset \vdash_{\mathsf{T}} A : \tau$ *?*

(ii) *Type Checking: for a given* τ, *is it true that* $\emptyset \vdash_{\mathsf{T}} A : \tau$ *?*

Proof

(i) *We give the sketch of a recursive algorithm (Figure 5) for building* τ *(or returning* false *if it does not exist).*

(ii) *We use the previous algorithm for type reconstruction (Figure 6). By uniqueness of typing, $\Gamma \vdash_{\mathsf{T}} A : \tau$ if and only if τ is equivalent to the type found for A.*

□

The next example shows that termination is not guaranteed for typable terms in RhoF.

Example 3.7 [Non Termination of Typable Terms [11]]
If $\Gamma \vdash_{\mathsf{T}} A : \tau$ then A can diverge. Take $\Gamma \triangleq f{:}(\iota \rightarrow \iota) \rightarrow \iota$, and $\Delta \triangleq X{:}\iota$, and $A \triangleq \omega\ f(\omega)$ with $\omega \triangleq f(X) \rightarrow_{\Delta} X\ f(X)$. Therefore, $\Gamma \vdash_{\mathsf{T}} \omega\ f(\omega) : \iota$, but $\omega\ f(\omega) \mapsto\!\!\!\to^{\mathsf{stk}}_{\rho\delta} \ldots$. This negative result proves that conjecture (i) of Exercise at pp. 14 of [10] was false. Notice that $\omega f(\omega)$ is typable without using the second-order features of RhoF.

4 The Polymorphic Type Inference uRhoF

In the previous section, we studied terms of the Rho-calculus decorated with types. In this *fully typed* approach, every closed term comes directly with a unique, intrinsic type. In this section, we discuss another way of giving types to terms of the Rho-calculus: the *type assignment* approach introduced by Curry [14] for the Theory of Combinators, and then modified by Curry and Feys [5,6]. The judgments have the shape $\Gamma \vdash_{\mathsf{U}} U : \tau$, where U is a term of the (untyped) Rho-calculus, τ is a type, and Γ is the context that assigns types to the free-variables of U and τ.

In this approach (called *à la* Curry by Barendregt), types are viewed as *predicates* (*properties*) of terms, and each closed term can be assigned either none or infinitely many types. Those systems are called *type assignment systems*. When we look at the Rho-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.

For the Lambda-calculus, in [14,23,21], it was observed that some of the type assignment systems already known in the literature can also be obtained from a fully typed system by means of an *erasing function* that erases type information from terms in a typed system. In particular, the Curry type assignment system ($F1$) [14] can be obtained from $\Lambda \rightarrow$, the polymorphic type assignment system ($F2$) [23] from $\Lambda 2$, and the higher-order type assignment

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{Type}^2(A;\Gamma) \triangleq \mathsf{match}\ A\ \mathsf{with}} \\
\alpha & \Rightarrow & * \\
 & & \mathsf{if}\ \alpha{:}* \in \Gamma \\
X/f & \Rightarrow & \tau \\
 & & \mathsf{if}\ X/f{:}\tau \in \Gamma \\
A_1\,;A_2 & \Rightarrow & \mathsf{Type}^2(A_1;\Gamma) \\
 & & \mathsf{if}\ \mathsf{Type}^2(A_1;\Gamma) = \mathsf{Type}^2(A_2;\Gamma) \\
P \mathrel{-\!\!\triangleright}_\Delta A_1 & \Rightarrow & \mathsf{Type}^2(P;\Gamma,\Delta) \mathrel{-\!\!\triangleright} \mathsf{Type}^2(A_1;\Gamma,\Delta) \\
 & & \mathsf{if}\ \mathsf{Type}^2(P;\Gamma,\Delta) \neq \mathsf{false}\ \neq \mathsf{Type}^2(A_1;\Gamma,\Delta) \\
 & & \mathsf{and}\ P \not\equiv \alpha \\
[P \ll_\Delta A_1]A_2 & \Rightarrow & \mathsf{Type}^2(A_2;\Gamma,\Delta) \\
 & & \mathsf{if}\ \mathsf{Type}^2(P;\Gamma,\Delta) = \mathsf{Type}^2(A_1;\Gamma,\Delta) \neq \mathsf{false} \\
 & & \mathsf{and}\ P \not\equiv \alpha \\
A_1\ A_2 & \Rightarrow & \tau_2 \\
 & & \mathsf{if}\ \mathsf{Type}^2(A_1;\Gamma) = \tau_1 \mathrel{-\!\!\triangleright} \tau_2\ \mathsf{and}\ \mathsf{Type}^2(A_2;\Gamma) = \tau_1 \\
\alpha \mathrel{-\!\!\triangleright}_* A_1 & \Rightarrow & \forall\alpha.\mathsf{Type}^2(A_1;\Gamma,\alpha{:}*) \\
 & & \mathsf{if}\ \mathsf{Type}^2(A_1;\Gamma,\alpha{:}*) \neq \mathsf{false} \\
[\alpha \ll_* \tau]A_1 & \Rightarrow & \mathsf{Type}^2(A_1;\Gamma,\alpha{:}*)\{\tau/\alpha\} \\
A_1\ \tau & \Rightarrow & \tau_1\{\tau/\alpha\} \\
 & & \mathsf{if}\ \mathsf{Type}^2(A_1;\Gamma) = \forall\alpha.\tau_1 \\
\_ & \Rightarrow & \mathsf{false}
\end{array}
$$

Figure 5. The Algorithm Type^2

system $(F\omega)$ [21] from the higher-order λ-calculus $\Lambda\omega$.

Let $\mathcal{D}er_{\mathsf{T}}$ be a typed derivation, and $(\!|-|\!)$ be the erasing function. By applying $(\!|-|\!)$ to the "subject" of every judgment in $\mathcal{D}er_{\mathsf{T}}$, we obtain a valid

$$\mathsf{Typecheck}^2(A;\Gamma;\tau) \triangleq \mathsf{if}\ \mathsf{Type}^2(A;\Gamma) = \tau\ \mathsf{then\ true\ else\ false}$$

Figure 6. The Algorithm $\mathsf{Typecheck}^2$

Syntactic Cat.	**Abstract Syntax**
As for RhoF	As for RhoF
$U.V.f \in \mathcal{T}erm$	$U.V ::= \mathsf{stk} \mid f \mid X \mid P \twoheadrightarrow U \mid [P \ll U]U \mid U\ U \mid U\,;U$

Figure 7. Syntax of uRhoF

type assignment derivation $\mathcal{D}er_{\mathsf{U}}$ with the same structure of the typed one. *Vice versa*, every type assignment derivation can be viewed as the result of an application of $(\!|-|\!)$ to a typed one. In particular, the erasing function $(\!|-|\!)$ induces an *isomorphism* between every typed system and the corresponding type assignment system.

Definition 4.1 The Erasing Function.

$$\begin{array}{lll|lll}
(\!|\mathsf{stk}|\!) & \triangleq & \mathsf{stk} & (\!|A\ \tau|\!) & \triangleq & (\!|A|\!) & \\
(\!|f|\!) & \triangleq & f & (\!|\alpha \twoheadrightarrow_* A|\!) & \triangleq & (\!|A|\!) & \\
(\!|X|\!) & \triangleq & X & (\!|[\alpha \ll_* \tau]B|\!) & \triangleq & (\!|B|\!) & \\
(\!|A\ B|\!) & \triangleq & (\!|A|\!)\ (\!|B|\!) & (\!|P \twoheadrightarrow_\Delta A|\!) & \triangleq & P \twoheadrightarrow (\!|A|\!) & P \not\equiv \alpha \\
(\!|A\,;B|\!) & \triangleq & (\!|A|\!)\,;(\!|B|\!) & (\!|[P \ll_\Delta A]B|\!) & \triangleq & [P \ll (\!|A|\!)](\!|B|\!) & P \not\equiv \alpha
\end{array}$$

This definition can easily be extended to derivations.

Syntax (Figure 7).

One can easily see that the syntax is obtained by simply "hiding" the types from the user. Type abstraction $(\alpha \twoheadrightarrow_* A)$ and type application $(A\ \tau)$ are no longer necessary since the polymorphism is fully implicit. As in ML, a term can be seen as an untyped one, but the typing machinery is called before accepting such a term.

Typing Rules (Figures 8 and 9).

A primitive polymorphic type assignment system was sketched in [10] (without any metatheory). It proves judgment of the shape:

$$\Gamma \vdash_{\mathsf{U}} ok \quad \text{and} \quad \Gamma \vdash_{\mathsf{U}} \tau : * \quad \text{and} \quad \Gamma \vdash_{\mathsf{U}} P : \tau \quad \text{and} \quad \Gamma \vdash_{\mathsf{U}} U : \tau$$

We discuss only the typing rules for well-formed terms and patterns which differ from the the corresponding typed ones.

Well-formed Contexts

$$\frac{}{\emptyset \vdash_{\mathsf{U}} ok}\,(Ctx{\cdot}Empty) \qquad \frac{\Gamma \vdash_{\mathsf{U}} ok \quad \alpha \notin \mathsf{Dom}(\Gamma)}{\Gamma, \alpha{:}{*} \vdash_{\mathsf{U}} ok}\,(Ctx{\cdot}Var^{\forall})$$

$$\frac{\Gamma \vdash_{\mathsf{U}} ok \quad \iota \notin \mathsf{Dom}(\Gamma)}{\Gamma, \iota{:}{*} \vdash_{\mathsf{U}} ok}\,(Ctx{\cdot}Const) \qquad \frac{\Gamma \vdash_{\mathsf{U}} ok \quad \Gamma \vdash_{\mathsf{U}} \tau : * \quad X \notin \mathsf{Dom}(\Gamma)}{\Gamma, X{:}\tau \vdash_{\mathsf{U}} ok}\,(Ctx{\cdot}Var)$$

Well-kinded Types

$$\frac{\Gamma_1, \iota{:}{*}, \Gamma_2 \vdash_{\mathsf{U}} ok}{\Gamma_1, \iota{:}{*}, \Gamma_2 \vdash_{\mathsf{U}} \iota : *}\,(Type{\cdot}Const) \qquad \frac{\Gamma_1, \alpha{:}{*}, \Gamma_2 \vdash_{\mathsf{U}} ok}{\Gamma_1, \alpha{:}{*}, \Gamma_2 \vdash_{\mathsf{U}} \alpha : *}\,(Type{\cdot}Var)$$

$$\frac{\Gamma, \alpha{:}{*} \vdash_{\mathsf{U}} \tau : *}{\Gamma \vdash_{\mathsf{U}} \forall\alpha.\tau : *}\,(Type{\cdot}Poly) \qquad \frac{\Gamma \vdash_{\mathsf{U}} \tau_1 : * \quad \Gamma \vdash_{\mathsf{U}} \tau_2 : *}{\Gamma \vdash_{\mathsf{U}} \tau_1 \rightarrow \tau_2 : *}\,(Type{\cdot}Arrow)$$

Figure 8. The Kind Assignment System for uRhoF

- $(Term{\cdot}Abs^{\rightarrow})$**:** The domain of Δ is given by the free-variables of P, *i.e.* $\mathsf{Dom}(\Delta) = \mathsf{Fv}(P)$.
- $(Term{\cdot}Abs^{\forall})$**:** This rule is not syntax directed; the classical side-condition about the freshness of α is enforced by the well-formedness of the context in the premises.
- $(Term{\cdot}Appl^{\forall})$**:** This rule is not syntax directed; the type τ_2 is guessed.
- $(Term{\cdot}Match^{\rightarrow})$**:** The context Δ is built from the free-variables of P, *i.e.* $\mathsf{Dom}(\Delta) = \mathsf{Fv}(P)$.

All the metaproperties presented below for uRhoF are adapted from system $F2$ of Leivant and for RhoF.

Theorem 4.2 (Subject Reduction for uRhoF)
If $\Gamma \vdash_{\mathsf{U}} U : \tau$ *and* $U \mapsto^{\mathsf{stk}}_{\rho\sigma\delta} V$, *then* $\Gamma \vdash_{\mathsf{U}} V : \tau$.

Proof *By an induction on the derivation of* $\Gamma \vdash_{\mathsf{U}} U : \tau$. □

Since uRhoF is essentially the counterpart of $F2$ of Leivant, and since Rho-calculus is a conservative extension of Lambda-calculus, it follows that type inference problem is undecidable.

Theorem 4.3 (Undecidability of Type Inference for uRhoF)
For a closed U *such that* $\mathsf{stk} \notin U$, *the following problem is undecidable:*

Well-formed Terms and Patterns

$$\frac{\Gamma \vdash_{\mathsf{U}} \tau : * \quad \tau \neq \bot}{\Gamma \vdash_{\mathsf{U}} \mathsf{stk} : \tau}\,(\mathit{Term{\cdot}Stuck})$$

$$\frac{\Gamma_1, X{:}\tau, \Gamma_2 \vdash_{\mathsf{U}} \mathit{ok}}{\Gamma_1, X{:}\tau, \Gamma_2 \vdash_{\mathsf{U}} X : \tau}\,(\mathit{Term{\cdot}Var}) \qquad \frac{\Gamma_1, f{:}\tau, \Gamma_2 \vdash_{\mathsf{U}} \mathit{ok}}{\Gamma_1, f{:}\tau, \Gamma_2 \vdash_{\mathsf{U}} f : \tau}\,(\mathit{Term{\cdot}Const})$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{U}} U : \tau_1 \rightarrowtriangle \tau_2\\ \Gamma \vdash_{\mathsf{U}} V : \tau_1\end{array}}{\Gamma \vdash_{\mathsf{U}} U\ V : \tau_2}\,(\mathit{Term{\cdot}Appl}^{\rightarrowtriangle}) \qquad \frac{\begin{array}{l}\Gamma, \Delta \vdash_{\mathsf{U}} P : \tau_1 \quad \mathsf{Dom}(\Delta) = \mathsf{Fv}(P)\\ \Gamma, \Delta \vdash_{\mathsf{U}} U : \tau_2 \quad \Gamma, \Delta \vdash_{\mathsf{U}} \tau_1 \rightarrowtriangle \tau_2 : *\end{array}}{\Gamma \vdash_{\mathsf{U}} P \rightarrowtriangle U : \tau_1 \rightarrowtriangle \tau_2}\,(\mathit{Term{\cdot}Abs}^{\rightarrowtriangle})$$

$$\frac{\Gamma, \alpha{:}* \vdash_{\mathsf{U}} U : \tau}{\Gamma \vdash_{\mathsf{U}} U : \forall\alpha.\tau}\,(\mathit{Term{\cdot}Abs}^{\forall}) \qquad \frac{\Gamma \vdash_{\mathsf{U}} U : \forall\alpha.\tau_1 \quad \Gamma \vdash_{\mathsf{U}} \tau_2 : *}{\Gamma \vdash_{\mathsf{U}} U : \tau_1\{\tau_2/\alpha\}}\,(\mathit{Term{\cdot}App}^{\forall})$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{U}} U : \tau\\ \Gamma \vdash_{\mathsf{U}} V : \tau\end{array}}{\Gamma \vdash_{\mathsf{U}} U\,;V : \tau}\,(\mathit{Term{\cdot}Struct}) \qquad \frac{\begin{array}{l}\Gamma, \Delta \vdash_{\mathsf{U}} P : \tau_1 \quad \mathsf{Dom}(\Delta) = \mathsf{Fv}(P)\\ \Gamma, \Delta \vdash_{\mathsf{U}} U : \tau_2 \quad \Gamma \vdash_{\mathsf{U}} V : \tau_1\end{array}}{\Gamma \vdash_{\mathsf{U}} [P \ll V]U : \tau_2}\,(\mathit{Term{\cdot}Match}^{\rightarrowtriangle})$$

Figure 9. The Type Assignment System for uRhoF

- *Type Inference: given Γ (gives meaning to constants), is there a type τ such that $\Gamma \vdash_{\mathsf{U}} U : \tau$?*

Proof *It follows* a fortiori *from the well known result of Wells [43].* □

5 RhoF *vs.* uRhoF *vs.* Logics

RhoF *vs.* uRhoF.

Writing or inferring types in programming languages is often a matter of taste; nice examples of flexible (often polymorphic) type-disciplines are ML and Haskell languages. The user can freely decorate his program with types or simply leave that job to the type inference module. The former choice gives to the user full-control on data-structures, while the latter delegates some choices to the type inference module. Both views are sound and operational. For example, since type-checking in RhoF is decidable, adding decidable polymorphic-types seems feasible and natural for Elan (since it is dynamically typed), and intriguing for Maude, because the latter has a sophisticated form of polytypic programming [4,16], where parameterized modules and theories can be defined and manipulated in a "nesting dolls" style (*matrioshka*).

From the point of view of type inference, the main motivation in introducing uRhoF is to find an easy way to validate *a posteriori* many existing lines of rewriting-based algorithms via static analysis.

The type inference module provides a safeguard to type-free code that is external rather than built-in. To be correctly applied, a type assignment system must enjoy fundamental properties, like the Church-Rosser property and the subject-reduction property. Normalization is not really an issue for kernel calculi underneath real programming languages. Therefore, to design a decidable version of uRhoF by customization of the well-known algorithm W of Damas-Milner [15] is an interesting challenge with practical consequences.

Logics.

The design of RhoF and uRhoF was also driven by a genuine interest in finding a suitable logic underneath both systems. The above mentioned erasing function $(\!| \cdot |\!)$, at least for RhoF and uRhoF, induces an isomorphism between the derivations in the corresponding systems. More precisely, if $\mathcal{D}er$ is a derivation in a typed system, by applying $(\!| \cdot |\!)$ to every object (*i.e.* term, constructor, or kind) in $\mathcal{D}er$, a valid derivation in the corresponding type assignment system is obtained. The other way around, every type assignment derivation can be obtained by applying $(\!| \cdot |\!)$ to a typed one.

Definition 5.1 [Isomorphism]
Let $Set(\mathcal{D}er_{\mathsf{T}})$ and $Set(\mathcal{D}er_{\mathsf{U}})$ be the sets of all derivations in RhoF and uRhoF. Systems RhoF and uRhoF are *isomorphic*, via $(\!| \cdot |\!)$, if and only if there are $\mathcal{F} : Set(\mathcal{D}er_{\mathsf{T}}) \Rightarrow Set(\mathcal{D}er_{\mathsf{U}})$ and $\mathcal{G} : Set(\mathcal{D}er_{\mathsf{U}}) \Rightarrow Set(\mathcal{D}er_{\mathsf{T}})$, such that:

(i) (Soundness) If $\mathcal{D}er_{\mathsf{T}} : \Gamma \vdash_{\mathsf{T}} A : \tau$, then $\mathcal{F}(\mathcal{D}er_{\mathsf{T}}) : (\!| \Gamma |\!) \vdash_{\mathsf{U}} (\!| A_t |\!) : \tau$.

(ii) (Completeness) If $\mathcal{D}er_{\mathsf{U}} : \Gamma' \vdash_{\mathsf{U}} U : \tau$, then $\mathcal{G}(\mathcal{D}er_{\mathsf{U}}) : \Gamma \vdash_{\mathsf{U}} A : \tau$, and $(\!| \Gamma |\!) = \Gamma'$, with $(\!| M |\!) = U$.

(iii) $\mathcal{F} \circ \mathcal{G}$ and $\mathcal{G} \circ \mathcal{F}$ are the identity on $Set(\mathcal{D}er_{\mathsf{U}})$ and $Set(\mathcal{D}er_{\mathsf{T}})$, respectively.

(iv) Both $\mathcal{F}$ and $\mathcal{G}$ preserve the structure of derivations, (*i.e.*, the tree obtained from a derivation by erasing all judgments, but not the names of the rules).

Notice that the definition of isomorphism expresses more than just soundness and completeness of $\mathcal{F}$. Indeed soundness and completeness imply an isomorphism between the judgments of the two systems, but they do not im-

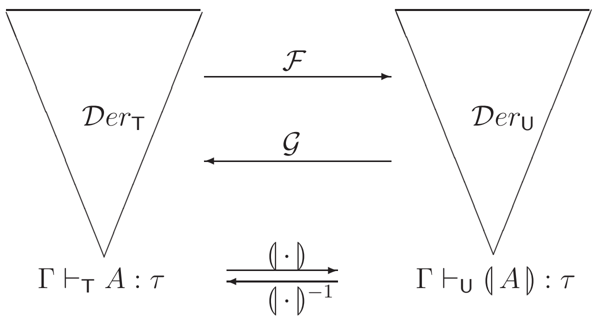
$\mathcal{F}$
$\mathcal{G}$
$\mathcal{D}er_{\mathsf{T}}$
$\mathcal{D}er_{\mathsf{U}}$
$\Gamma \vdash_{\mathsf{T}} A : \tau$
$\Gamma \vdash_{\mathsf{U}} (\!| A |\!) : \tau$
$(\!| \cdot |\!)$
$(\!| \cdot |\!)^{-1}$

Figure 10. Functions between Typed and Untyped Judgments and Derivations.

ply necessarily a one-to-one correspondence between proofs.

Functions $\mathcal{F}$ and $\mathcal{G}$ are omitted; their constructions are left as an easy exercise to the interested reader. Figure 10 depicts the various functions between typed and untyped systems of Rho-calculus that realize the above relations between typed and untyped judgments and derivations. Similar functions for the Lambda-calculus, relating typed system *à la* Church with corresponding type assignments systems *à la* Curry can be found in [24,28].

Theorem 5.2 (RhoF *vs.* uRhoF)
Systems RhoF, and uRhoF are isomorphic.

Proof *The proof follows the same lines of [18].* □

The relation with logic through the so-called Curry-Howard isomorphism [22], or *"formulæ-as-types and proofs-as-terms"* principle, has been deeply studied for the Lambda-calculus. As demonstrated in [9,3], this relation is not so clear for the Rho-calculus. The principle could be adapted for RhoF as follows:

> *Given a typed term A, if we can derive for A a type τ in the typed system RhoF, with a derivation $\mathcal{D}er_{\mathsf{T}}$, then the term A can be seen as the coding of a logical proof, proving the formula φ that can be interpreted as the type τ assigned to A.*

For the typed system RhoF, the current issue is to find a suitable logic that fits with the calculus. If patterns are simple variables, the logic is minimal second-order propositional logic, but in the case of more complicated patterns, much work has to be done.

For the type assignment system uRhoF the relation with logic is not so clear even for the corresponding type assignments for the Lambda-calculus. The 'formulæ-as-types' principle of Curry and Howard can be extended to the above type assignment systems as follows [24]:

Given an untyped term U, if we can assign a type τ in the type assignment system uRhoF*, with a derivation $\mathcal{D}er_{\mathsf{U}}$, then:*

- *$\mathcal{D}er_{\mathsf{U}}$ can be interpreted as the coding of a proof for the logic formulas φ which corresponds to the interpretation of the type τ assigned to U;*
- *U can be interpreted as the coding of a "logical proof schemas", whose instances (of the schema) prove, respectively, all the logic formulas φ_i's that can be interpreted as the types τ_i's that can be assigned to U.*

Clearly, the fact that the classes of derivations for the typed system and the type assignment system are isomorphic means that they have the same underlying logical system.

Logical (In)Consistency.

One major requirement when associating a type system with a logic is consistency, *i.e.* the impossibility to write a closed term with type *falsum*; the term below seems to "spoof" consistency at the price of only one algebraic constant f declared in the context.

Example 5.3 [Logical Inconsistency]
Let $\Gamma \triangleq f{:}\bot \to \bot \to \bot$.

$$\dfrac{\dfrac{\dfrac{\begin{array}{l}\Gamma, X{:}\bot \vdash_{\mathsf{T}} X : \bot \\ \Gamma, X{:}\bot \vdash_{\mathsf{T}} f : \bot \to \bot \to \bot\end{array}}{\Gamma, X{:}\bot \vdash_{\mathsf{T}} f(X) : \bot \to \bot} \qquad \Gamma, X{:}\bot \vdash_{\mathsf{T}} X : \bot}{\Gamma \vdash_{\mathsf{T}} f(X) \to_{(X:\bot)} X : (\bot \to \bot) \to \bot} \qquad \dfrac{}{\Gamma \vdash_{\mathsf{T}} X \to_{\bot} X : \bot \to \bot}}{\Gamma \vdash_{\mathsf{T}} (f(X) \to_{(X:\bot)} X)\ (X \to_{\bot} X) : \bot}$$

The key point of this spoofer seems to be the contravariant position of $\bot$ in the typing of f: this allows to inhabit falsum, that is, prove the inconsistency of the system. This is not surprising, since our application rule hides in the first premise a "logical-cut", *e.g.* here the assumption $X : \bot$ is forgotten.

Recovering Consistency.

Below, we propose a well-known restriction, due to Mendler [25] that would block the counterexample to the Strong Normalization (Example 3.7), and the counterexample to the Logical Consistency (Example 5.3). The consistency of the system is guaranteed by shrinking the set of typable terms. Mendler has shown that, when introducing recursive definitions in the typed Lambda-calculus, strong normalization is no longer enforced by typing if the type constructors do not satisfy a *"positiveness condition"*. This kind of condition is still present in the Calculus of Inductive Constructions which is the basis of the Coq proof assistant. The issue appears in programming languages too: for instance, in ML, one can define any recursive function without using the keyword let rec.

As simple modification in both RhoF and uRhoF can restrict the $(Type{\cdot}Arrow)$ as follows:

$$\frac{\Gamma \vdash_{\mathsf{T}} \tau_1 : * \quad \Gamma \vdash_{\mathsf{T}} \tau_2 : * \quad \text{if } \tau_2 \equiv \tau_2^1 \twoheadrightarrow \ldots \tau_2^n \twoheadrightarrow \alpha \text{ then } \alpha > 0 \text{ in } \tau_1}{\Gamma \vdash_{\mathsf{T}} \tau_1 \twoheadrightarrow \tau_2 : *}\,(Type{\cdot}Arrow')$$

where $\alpha > 0$ in τ_1 if:

(i) α does not occur in τ_1;

(ii) or $\tau_1 = \tau_1^1 \twoheadrightarrow \tau_1^2$ where α does not occur in τ_1^1 and $\alpha > 0$ in τ_1^2.

This positiveness condition is the price to pay in order to have a sound logical system.

Conjecture 5.4 (Consistency)
For RhoF and uRhoF (with the rule $(Type{\cdot}Arrow')$) and every closed A and U, and for a suitable Γ giving meaning to the algebraic constants of A and U, respectively, the following holds: $\Gamma \not\vdash_{\mathsf{T}} A : \bot$, and $\Gamma \not\vdash_{\mathsf{U}} U : \bot$.

The above conjecture left open another conjecture about the existence of a powerful logic that checks the shape of the proofs/terms (inhabitants of formulæ/types) via pattern-matching before applying a cut/application rule.

Conjecture 5.5 (Polymorphic Rho-gic) *We conjecture that there exists a logical system underneath RhoF and uRhoF based on pattern-matching.*

It would be worthy to explore Meseguer's *Conditional Rewriting Logic* [26].

6 Related Work and Conclusions

In this paper we presented two systems, the Fully-typed Polymorphic Rho-calculus (RhoF) and the Type Inference Polymorphic Rho-calculus (uRhoF): both systems enjoy subject reduction of typable terms. RhoF also enjoys the decidability of type checking and of type reconstruction.

Because of the decidability of type-checking in RhoF, customizing an existing rewriting-language with polymorphic-types seems an interesting alternative to validate code without limiting code expressiveness. From the point of view of type inference, the main motivation, in introducing uRhoF, is to find an easy way to validate code of many existing lines of rewriting-based algorithms via static analysis.

We also discussed the problematic of finding a suitable consistent logic that corresponds to the two systems via the Curry-Howard isomorphism.

The next question in our agenda is to study a variant of uRhoF (called $\mathsf{uRhoF}_{\mathsf{mlet}}$) featuring a restricted form of polymorphism *à la* Damas-Milner-Tofte and to customize of the well-known algorithm W of Damas-Milner [15], conjecture (ii) of Exercise pp. 14 in [10].

Acknowledgments.

The authors are sincerely grateful to Claude Kirchner for many fruitful discussions and invaluable comments about this work, and the anonymous referees for their extremely useful comments.

References

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

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

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

[4] M. Clavel, F. Durán, and N. Martí-Oliet. Polytypic Programming in Maude. In *Proc. of WRLA*. ENTCS, 2000.

[5] H.B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

[6] H. B. Curry, J. R. Hindley, and J. P. Seldin. *Combinatory Logic*, volume 2. North-Holland, Amsterdam, 1972.

[7] H. Cirstea. *Rewriting Calculus: Foundations and Applications*. PhD thesis, Université Henri Poincaré - Nancy I, 2000.

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

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

[10] H. Cirstea, C. Kirchner, and L. Liquori. Rewriting Calculus with(out) Types. In *Proc. of WRLA*, volume 71 of *ENTCS*, 2002.

[11] H. Cirstea, C. Kirchner, L. Liquori, and B. Wack. The Rho Cube : Some Results, Some Problems. In *Proc. of HOR*, 2002. Also as LORIA Research Report A02-R-470.

[12] H. Cirstea, L. Liquori, and B. Wack. Rho-calculus with Fixpoint: First-order System. In *Proc. of TYPES*. Springer-Verlag, 2004.

[13] M. Clavel and J. Meseguer. Reflection in Conditional Rewriting Logic. *Theoretical Computer Science*, 285(2):245–288, 2002.

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

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

[16] F. Durán and J. Meseguer. Parameterized Theories and Views in Full Maude 2.0. In *Proc. of WRLA*. ENTCS, 2000.

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

[18] P. Giannini, F. Honsell, and S. Ronchi della Rocca. Type Inference: Some Results, Some Problems. *Fundamenta Informaticae*, 19((1,2)):87–126, 1993.

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

[20] J. Goguen. The OBJ Family Home Page, 2004. http://www.cs.ucsd.edu/users/goguen/sys/obj.html.

[21] 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.

[22] W. Howard. The formulas–as–types notion of construction. In *Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980.

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

[24] L. Liquori. *Type Assigment Systems for Lambda Calculi and for the Lambda Calculus of Objects*. PhD thesis, University of Turin, 1996.

[25] N. P. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, USA, 1987.

[26] José Meseguer. Conditional Rewriting Logic: an Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[27] Microsoft. The C# Home Page, 2004. http://msdn.microsoft.com/vcsharp/.

[28] A. Miquel. *Le Calcul des Constructions Implicite: Syntaxe et Sémantique*. PhD thesis, Université Denis Diderot, Paris 7, 2001.

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

[30] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

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

[32] 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.

[33] Sun. Java Technology, 2004. http://java.sun.com/.

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

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

[36] The Asf+Sdf Team. The Asf+Sdf Meta-Environment Home Page, 2004. http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/MetaEnvironment.

[37] The Haskell Team. The Haskell Home Page, 2004. `http://www.haskell.org/`.

[38] The Maude Team. The Maude Home Page, 2004. `http://maude.cs.uiuc.edu/`.

[39] The Protheo Team. The Elan Home Page, 2004. `http://elan.loria.fr`.

[40] The Scheme Team. The Scheme Language, 2004. `http://www.swiss.ai.mit.edu/projects/scheme/`.

[41] A. van Deursen, J. Heering, and P. Klint. *Language Prototyping*. World Scientific, 1996.

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

[43] 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.