The Polymorphic Rewriting-calculus

[Type Checking vs. Type Inference]

Luigi Liquori\textsuperscript{a} and Benjamin Wack\textsuperscript{b}

\textsuperscript{a} INRIA Sophia Antipolis
\textsuperscript{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 (Rho\textsuperscript{F}) that enjoys subject reduction, type uniqueness, and decidability of typing. We then apply a classical type-erasing function to Rho\textsuperscript{F} obtaining an untyped Rho-calculus à la Curry (uRho\textsuperscript{F}). The related type inference system is isomorphic to Rho\textsuperscript{F} and enjoys subject reduction. Both Rho\textsuperscript{F} and uRho\textsuperscript{F} 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.

Key words: 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 [MOM02] (Elan[The04d], Maude[The04c], ASF+SDF[vDHK96,The04a], OBJ*[FN96,Gog04], . . .). Although these languages are less used than object-oriented languages (Java[Sun04], C#[Mic04], . . .), 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 in 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, \textit{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\cite{MTHM97,The03a}, Haskell\cite{The04b}, Scheme\cite{The04c}, or Prolog\cite{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 an \textit{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 \cite{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 \cite{Ste02}.

The \textit{Rewriting-calculus} \cite{CKL01b,CLW04} 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 $P \rightarrow A$, 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 $P \rightarrow A$ 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 more elaborated equational theories.

A large class of Term Rewriting Systems are also embedded in the Rewriting-calculus. In particular, the notions of \textit{rule application} and \textit{result} (basic ingredients of Term Rewriting Systems) become explicit. A rewrite rule becomes 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 \textit{implicit} as in ASF+SDF\cite{vDHK96}, \textit{local} as in OBJ* and Maude, or \textit{user defined} as in Elan and Maude\cite{CM02}. Previous papers \cite{Cir00,CKL01a,CLW04} showed that the Rewriting-calculus can be used as an core engine calculus for rewriting-based languages such as Elan and Maude. In particular, it could be intriguing to compare/merge/interleave the \textit{core engine}.

Finally, \textit{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. We then apply a classical erasing function to RhoF in order to obtain a corre-
sponding 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 bounded 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-guided, 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 “\_ → \_” (function- and type-abstraction), and “[\_ ≪ \_]” (delayed matching constraint), and “\_ ; \_” (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 “[ ≪ ]” which is higher than that of “→” which is, in turn, of higher priority than the “;”. The symbol τ ranges over the set Type of types, the symbol ι ranges over the set Type_K of type constants (Type_K ⊆ Type), the symbols α, β range over the set Type_V of type-variables (Type_V ⊆ Type), the symbols A, B, C, . . . , U, V, W, Z range over the set Term of (un)typed terms, the symbols X, Y, Z, . . . range over the set Var of term variables (Var ⊆ Term),
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 \([\nu090]\) is needed to keep the small-step semantics confluent. A typed rewriting rule of the form \( P \rightarrow_{\Delta} A \) abstracting over the free-variables of \( P \) is a first-class citizen of the calculus. The context \( \Delta \) records the type of the free-variables of \( P \) (bound in \( A \)). When a pattern is a variable, we write \( X \rightarrow_{\tau} A \), instead of \( X \rightarrow_{\phi(X,\tau)} A \) (by a little abuse of notation). An application is implicitly denoted by the (hidden) symbol “\( \cdot \)”; 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 \( \Delta \)) constrained by the matching between \( P \) and \( A \). The symbol \( \text{stk} \) can be considered as 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 \( \text{Fv} \)]

\[
\begin{align*}
\text{Fv}(f) & \triangleq \emptyset & \text{Fv}(P \rightarrow_\Delta A) & \triangleq \text{Fv}(A) \cup \text{Fv}(\Delta) \setminus \text{Fv}(P) \\
\text{Fv}(\text{stk}) & \triangleq \emptyset & \text{Fv}([P \ll_\Delta A]B) & \triangleq \text{Fv}((P;\Delta \rightarrow B) A) \\
\text{Fv}(X) & \triangleq \{X\} & \text{Fv}(A; B/A \ B) & \triangleq \text{Fv}(A) \cup \text{Fv}(B) \\
\text{Fv}(\alpha) & \triangleq \{\alpha\} & \text{Fv}(\alpha \tau) & \triangleq \text{Fv}(A) \cup \text{Fv}(\tau)
\end{align*}
\]

As usual, we work modulo \( \alpha \)-conversion and we adopt Barendregt’s “hygiene-convention” [Bar84], 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 \( \theta \) is a mapping from the set of term (resp. type) variables to the set of terms resp. types. A finite substitution \( \theta \) 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 is \( \{X_1, \ldots, X_m\} \), or \( \{\alpha_1, \ldots, \alpha_m\} \) is denoted by \( \text{Dom}(\theta) \). The application of a substitution \( \theta \) to a term \( A \) (resp. type \( \tau \)), denoted by \( A\theta \) (resp. \( \tau\theta \)), is defined as follows:

\[
\begin{align*}
f\theta & \triangleq f & (P \rightarrow_\Delta A)\theta & \triangleq P \rightarrow_\Delta A\theta \\
\text{stk}\theta & \triangleq \text{stk} & ([P \ll_\Delta A]B)\theta & \triangleq [P \ll_\Delta A\theta]B\theta \\
(A \tau)\theta & \triangleq (A\theta) (\tau\theta) & (A; B/A \ B)\theta & \triangleq A\theta; B\theta/A\theta \ B\theta \\
X_i\theta & \begin{cases} A_i & \text{if } X_i \in \text{Dom}(\theta) \\ X_i & \text{otherwise} \end{cases} & \alpha_i\theta & \begin{cases} \tau_i & \text{if } \alpha_i \in \text{Dom}(\theta) \\ \alpha_i & \text{otherwise} \end{cases} \\
\iota\theta & \triangleq \iota & (\tau_1 \rightarrow_\Delta \tau_2)\theta & \triangleq \tau_1\theta \rightarrow \tau_2\theta
\end{align*}
\]

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 [CLW04]) 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 \( T \)
A matching equation is a problem $T \overset{\Delta}{\leftarrow} P \leftarrow_T A$ where $P$ is a pattern and $A$ is a term;

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

Different theories and the corresponding pattern-matching problems can be formally defined and solved, for example as explained in [CKL01a]. If the equation $P \overset{\Delta}{\leftarrow} T A$ has a unique solution, we denote it by $\theta(P \overset{\Delta}{\leftarrow} T A)$.

We define a superposition relation $\sqsubseteq: \text{Pattern} \times \text{Term}$ 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:

\[
\begin{align*}
  f &\quad \sqsubseteq \quad f \\
  \text{stk} &\quad \sqsubseteq \quad \text{stk} \\
  X &\quad \sqsubseteq \quad A \quad (\forall A) \\
  \alpha &\quad \sqsubseteq \quad \tau \quad (\forall \tau)
\end{align*}
\]

\[
\begin{align*}
  f(P) &\quad \sqsubseteq \quad A \quad \text{if } A \equiv f(B) \land \bar{P} \sqsubseteq \bar{B} \\
  P &\quad \sqsubseteq \quad A \quad \text{if } A \equiv
  \begin{cases}
    X \lor (A_1; A_2) \lor A \lor \tau & (\forall A) \\
    (A_1, A_2 \land A_1 \notin \text{Pattern}) \lor (\forall P)
  \end{cases}
\end{align*}
\]

(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, $T_{\text{stk}}$]**
The relation $\rightarrow_{\text{stk}}$ is defined by the following rules:

\[
\begin{align*}
  [P \ll_A B] &\quad \rightarrow_{\text{stk}} \quad \text{stk} \quad \text{if } P \not\sqsubseteq A \\
  \text{stk}; A &\quad \rightarrow_{\text{stk}} \quad \text{stk} \\
  A; \text{stk} &\quad \rightarrow_{\text{stk}} \quad \text{stk} \\
  \text{stk} A &\quad \rightarrow_{\text{stk}} \quad \text{stk}
\end{align*}
\]

We denote by $\twoheadrightarrow_{\text{stk}}$ the contextual closure induced by these rules. Its reflexive and transitive closure is denoted by $\twoheadrightarrow_{\text{st}}$. The symmetric and transitive closure of $\twoheadrightarrow_{\text{stk}}$ is denoted by $=_{\text{stk}}$. Let $T_{\text{stk}}$ be the theory associated to the congruence $=_{\text{stk}}$. Matching equations in the theory $T_{\text{stk}}$ are denoted $P \ll_{\text{stk}} A$. 
\[(P \rightarrow_{\Delta} A) \ B \rightarrow_{\rho} [P \triangleleft_{\Delta} B] A\]

\[[P \triangleleft_{\Delta} B] A \rightarrow_{\sigma} A\theta(P \triangleleft_{\stk} B)\]

\[(A; B) \ C \rightarrow_{\delta} A \ C; B \ C\]

\[A \rightarrow_{\stk} B\]

Figure 2. Top-level Rules of RhoF

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 \(T_{\stk}\) (recall the symbols \(A, B, C\) range over \(Type \cup Term\)).

Let us quickly explain the top-level rules:

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

\((\sigma)\) this rule is applied if and only if the matching equation \(P \triangleleft_{\stk} B\) 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 \(B\) so that the equation has a solution and the rule can be triggered.

\((\delta)\) 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 \(T_{\stk}\)-theory.

We denote by \(\rightarrow_{\rho}\) the contextual closure induced by these rules. Its reflexive and transitive closure is denoted by \(\rightarrow_{\rho}^{\ast}\). The symmetric and transitive closure of \(\rightarrow_{\rho}\) is denoted by \(=_{\rho}\). Notice that these relations are parameterized over the adopted theory \(T_{\stk}\). We denote by \(\rightarrow_{\stk}^{\ast}\) the relation \(\rightarrow_{\stk} \cup \rightarrow_{\rho}^{\ast}\). For \(\rightarrow_{\rho}^{\ast}\), the following holds.

**Theorem 2.6 (Church Rosser for RhoF [CLW04])**

The relation \(\rightarrow_{\stk}^{\ast}\) is confluent.
Well-formed contexts

\[ \emptyset \vdash ok \]  \hspace{1cm} \Gamma \vdash ok \not\in \text{Dom}(\Gamma) \quad (\text{Ctx-Empty})  \\
\Gamma \vdash ok \hspace{1cm} \Gamma, \iota:\ast \vdash ok \] \hspace{1cm} \Gamma \vdash \tau : \ast \not\in \text{Dom}(\Gamma) \quad (\text{Ctx-Const})  \\
\Gamma, \iota:\ast \vdash ok \] \hspace{1cm} \Gamma, X:\tau \vdash ok \quad (\text{Ctx-Var})

Well-kinded Types

\[ \Gamma_1, \iota:\ast, \Gamma_2 \vdash ok \] \hspace{1cm} \Gamma_1, \alpha:\ast, \Gamma_2 \vdash ok \] \hspace{1cm} \Gamma_1, \alpha:\ast, \Gamma_2 \vdash \alpha : \ast \quad (\text{Type-Const})  \\
\Gamma, \alpha:\ast \vdash \tau : \ast \] \hspace{1cm} \Gamma \vdash \forall \alpha. \tau : \ast \quad (\text{Type-Poly})  \\
\Gamma \vdash \tau_1 : \ast \hspace{1cm} \Gamma \vdash \tau_2 : \ast \quad (\text{Type-Arrow})

Figure 3. The Kind System for RhoF

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 $\vdash_T$ (for Typed terms). A typed system is a set of rules for proving judgments of the shape $\Gamma \vdash_T A : \tau$, where $A$ is a typed term, $\tau$ is a type, and $\Gamma$ is a context. The meaning of such a judgment is: the term $A$ has type $\tau$ under the context $\Gamma$, and $\Gamma$ records the types of the free-variables of $\Gamma$ and $\tau$. Figures 3, and 4 presents the kinding/typing rules of RhoF, which are directly inspired by the Girard System F [Gir86]. More precisely, the system proves judgment of the shape:

$\Gamma \vdash_T ok$ and $\Gamma \vdash_T \tau : \ast$ and $\Gamma \vdash_T P : \tau$ and $\Gamma \vdash_T A : \tau$

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

- *(Term-Var)(Term-Const)*: As usual, the context determines the type of variables. It cannot contain two declarations for the same variable (or constant);

- *(Term-Stuck)*: Since stk can appear in any structure, its type can be virtually anything but falsum, i.e. $\perp \equiv \forall \alpha. \alpha$;

- *(Term-Abs-*)*: 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. $\tau_2$ disappears in the final type of $f(X)$
Well-formed Terms and Patterns

\[ \frac{\Gamma_1, X : \tau, \Gamma_2 \vdash \text{ok} \quad \Gamma_1, f : \tau, \Gamma_2 \vdash \text{ok}}{\Gamma_1, X : \tau, \Gamma_2 \vdash \text{ok}} \]

\[ \frac{\Gamma_1, \Delta \vdash T : \tau}{\Gamma_1, \Delta \vdash f : \tau} \]

\[ \frac{\Gamma \vdash A : \tau_1 \rightarrow \tau_2}{\Gamma \vdash \lambda \Delta : \tau_2 \vdash \tau_1 \rightarrow \tau_2} \]

\[ \frac{\Gamma \vdash B : \tau_1}{\Gamma \vdash \lambda \Delta : \tau_2 \vdash \tau_1 \rightarrow \tau_2} \]

\[ \frac{\Gamma, \Delta \vdash P : \tau_1}{\Gamma, \Delta \vdash P : \tau_1} \]

\[ \frac{\Gamma, \Delta \vdash A : \tau_2}{\Gamma, \Delta \vdash \lambda \Delta : \tau_2 \vdash \tau_1 \rightarrow \tau_2} \]

\[ \frac{\Gamma \vdash * \rightarrow A : \tau}{\Gamma \vdash \lambda \alpha : \tau \vdash A : \forall \alpha \cdot \tau} \]

\[ \frac{\Gamma \vdash A : \tau}{\Gamma \vdash \lambda \alpha : \tau \vdash A : \forall \alpha \cdot \tau} \]

\[ \frac{\Gamma \vdash \lambda \alpha : \tau \rightarrow A : \forall \alpha \cdot \tau}{\Gamma \vdash \lambda \alpha : \tau \vdash A : \forall \alpha \cdot \tau} \]

\[ \frac{\Gamma \vdash A : \tau \rightarrow \tau_2 \neq \bot}{\Gamma \vdash \lambda \Delta : \tau \vdash \tau_2} \]

\[ \frac{\Gamma \vdash \lambda \alpha : \tau \rightarrow A : \forall \alpha \cdot \tau}{\Gamma \vdash \lambda \alpha : \tau \vdash A : \forall \alpha \cdot \tau} \]

in the judgment \( f : \tau_2 \rightarrow \tau_1, X : \tau_2 \vdash f(X) : \tau_1 \). The context \( \Delta \) gives the types of the free-variables of \( P \). The type system ensures that the solutions of the corresponding matching equations are well-typed;

- (Term-Appl): We directly exploit the information given in the type of the function, statically checking that the given argument has the expected type \( \tau_1 \);

- (Term-Abs\( ^\beta \)): The rationale is: \( \alpha \rightarrow * A \simeq \alpha \rightarrow (\alpha : \tau_1) A \). Abstraction on type-variables makes the polymorphic mechanism available at the user-level: note that a trivial pattern is used in polymorphic-abstraction.

- (Term-Appl\( ^\nu \)): The rationale is: all free occurrences of \( \alpha \) in \( \tau_1 \) are substituted with \( \tau_2 \). Any well-formed type \( \tau_2 \) is suitable, which makes the typing fully polymorphic.

- (Term-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-Match\( ^\rightarrow \)) (Term-Match\( ^\right\)): This rule states that the constraint \( [P \right\Delta B] A \) gets the same type as \( (P \rightarrow \Delta A) B \). This is sound since \( (P \rightarrow \Delta A) B \rightarrow \rho \)
Example 3.1 [Some derivable typing judgments [Bar92]-inspired] Let \( \Gamma \triangleq \iota : \star, f \rightarrow \iota, a : \iota \). The following judgments are derivable:

\[
\begin{align*}
\emptyset \vdash T \perp & \equiv \forall \alpha. (\perp \rightarrow \alpha) & \text{Second-order definition of falsum} \\
\emptyset \vdash \alpha \rightarrow \perp & \rightarrow (X \alpha) : \forall \alpha. (\perp \rightarrow \alpha) & \text{Ex falso sequitur quodlibet}^1 \\
\emptyset \vdash \beta \rightarrow \perp & \rightarrow (\beta \rightarrow \beta) & \text{Polymorphic identity} \\
\Gamma \vdash (\gamma \rightarrow \perp) f(Z) \rightarrow (Z: \gamma) Z : f(a) : \iota & \text{Polymorphic instantiation-application}
\end{align*}
\]

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

Remark 3.2 [Spoof [BCKL03]] If we allow variables as the head symbol of a pattern (called “active variables”), then we can write the following counterexample. In the context \( \Gamma \triangleq X : \tau_1 \rightarrow \tau_2, Y : \tau_1, f : \tau_3 \rightarrow \tau_2, a : \tau_3 \), the pattern

\[
\begin{align*}
\Gamma \vdash X(Y) : \tau_2 & \quad \text{and the term} \quad \Gamma \vdash f(a) : \tau_2
\end{align*}
\]

have a common type, but the solution of the matching problem \( X(Y) \not\prec f(a) \) instantiates \( X \) and \( Y \) with terms not having the expected type (i.e., subject reduction is lost). If \( \Gamma \triangleq f : \tau_3 \rightarrow \tau_2, a : \tau_3 \) and \( \Delta \triangleq X : \tau_1 \rightarrow \tau_2, Y : \tau_1 \), then \( \Gamma \vdash X(Y) \not\prec \Delta Y f(a) : \tau_1 \) but \( \Gamma \vdash a : \tau_3 \).

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 \( \Gamma, \Delta \vdash P : \tau \) and \( \Gamma \vdash B : \tau \) and \( \text{Dom}(\Delta) = \text{Fv}(P) \), are such that \( P \not\prec_{\text{stk}} B \) has a solution \( \theta \), then for all \( X \in \text{Fv}(P) \), there exists \( \sigma \) such that \( \Gamma, \Delta \vdash X : \sigma \) and \( \Gamma \vdash X \theta : \sigma \).

(ii) If \( \Gamma, \Delta \vdash A : \tau \), then for any well-typed substitution \( \theta \) such that \( \text{Dom}(\theta) = \text{Dom}(\Delta) \), we have \( \Gamma \vdash A \theta : \tau \).

\[^1\] Anything follows from a false judgment: the subject of this judgment is its proof.
Liquori and Wack

Theorem 3.4 (Subject Reduction for RhoF)
If \( \Gamma \vdash T A : \tau \) and \( A \mapsto \text{stk} B \), then \( \Gamma \vdash T B : \tau \).

**Proof** By an induction on the derivation of \( \Gamma \vdash T A : \tau \).

Theorem 3.5 (Type Uniqueness for \( \rhoF_\text{fix} \))
If \( \Gamma \vdash T A : \tau_1 \) and \( \Gamma \vdash T A : \tau_2 \), and \( \text{stk} \notin A \), 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 \( \text{stk} \notin A \), the following problems are decidable:

(i) Type Reconstruction: is there a type \( \tau \) such that \( \emptyset \vdash T A : \tau \) ?

(ii) Type Checking: for a given \( \tau \), is it true that \( \emptyset \vdash T A : \tau \) ?

**Proof**

(i) We give the sketch of a recursive algorithm (Figure 5) for building \( \tau \) (or returning \( \text{false} \) if it does not exist).

(ii) We use the previous algorithm for type reconstruction (Figure 6). By uniqueness of typing, \( \Gamma \vdash T A : \tau \) if and only if \( \tau \) 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 [CKLW02]]
If \( \Gamma \vdash T A : \tau \) then \( A \) can diverge. Take \( \Gamma \triangleleft f : \iota \to \iota \triangleleft \iota \), and \( \Delta \triangleleft X : \iota \), and \( A \triangleleft \omega f(\omega) \) with \( \omega \triangleleft f(\omega) \to \Delta X f(X) \). Therefore, \( \Gamma \vdash T \omega f(\omega) : \iota \), but \( \omega f(\omega) \mapsto \text{stk} \ldots \). This negative result proves that conjecture (i) of Exercise at pp. 14 of [CKL02] 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 [Cur34] for the Theory of Combinators, and then modified by Curry and Feys [CF58,CHS72]. The judgments have the shape \( \Gamma \vdash_U U : \tau \), where \( U \) is a term of the (untyped) Rho-calculus, \( \tau \) is a type, and \( \Gamma \) is the context that assigns types to the free-variables of \( U \) and \( \tau \).

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
Type\(^2(A; \Gamma) \triangleq \) match \(A\) with
\[
\begin{align*}
\alpha \Rightarrow * & \quad \text{if } \alpha:* \in \Gamma \\
X/f \Rightarrow \tau & \quad \text{if } X/f: \tau \in \Gamma \\
A_1; A_2 \Rightarrow \text{Type}\(^2\)(A_1; \Gamma) & \quad \text{if Type}\(^2\)(A_1; \Gamma) \equiv \text{Type}\(^2\)(A_2; \Gamma) \\
P \gets_\Delta A_1 \Rightarrow \text{Type}\(^2\)(P; \Gamma, \Delta) \gets \text{Type}\(^2\)(A_1; \Gamma, \Delta) & \quad \text{if Type}\(^2\)(P; \Gamma, \Delta) \neq \text{false} \neq \text{Type}\(^2\)(A_1; \Gamma, \Delta) \quad \text{and } P \neq \alpha \\
[P \ll_\Delta A_1]A_2 \Rightarrow \text{Type}\(^2\)(A_2; \Gamma, \Delta) & \quad \text{if Type}\(^2\)(P; \Gamma, \Delta) \equiv \text{Type}\(^2\)(A_1; \Gamma, \Delta) \neq \text{false} \quad \text{and } P \neq \alpha \\
A_1 A_2 \Rightarrow \tau_2 & \quad \text{if Type}\(^2\)(A_1; \Gamma) \equiv \tau_1 \gets \tau_2 \text{ and Type}\(^2\)(A_2; \Gamma) \equiv \tau_1 \\
\alpha:* \gets A_1 \Rightarrow \forall \alpha. \text{Type}\(^2\)(A_1; \Gamma, \alpha:* ) & \quad \text{if Type}\(^2\)(A_1; \Gamma, \alpha:* ) \neq \text{false} \\
[\alpha \ll_* \tau]A_1 \Rightarrow \text{Type}\(^2\)(A_1; \Gamma, \alpha:* )\{\tau/\alpha\} & \\
A_1 \tau \Rightarrow \tau_1\{\tau/\alpha\} & \quad \text{if Type}\(^2\)(A_1; \Gamma) \equiv \forall \alpha. \tau_1 \\
\_ \Rightarrow \text{false} &
\end{align*}
\]

Figure 5. The Algorithm Type\(^2\)

Typecheck\(^2\)(A; \Gamma; \tau_1) \triangleq \tau_2 := \text{Type}\(^2\)(A; \Gamma) \\
if \tau_2 \equiv \text{false} \text{ then return false else return } (\tau_1 = \tau_2)

Figure 6. The Algorithm Typecheck\(^2\)

none or infinitely many types. Those systems are called type assignment sys-
Liquori and Wack

tems. 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 compile-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 [Cur34, Lei83, GR88], 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) [Cur34] can be obtained from Λ→, the polymorphic type assignment system (F2) [Lei83] from Λ2, and the higher-order type assignment system (Fω) [GR88] from the higher-order λ-calculus Λω.

Let DerT be a typed derivation, and 〈−〉 be the erasing function. By applying 〈−〉 to the “subject” of every judgment in DerT, we obtain a valid type assignment derivation DerU 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.

\[
\begin{align*}
\langle \text{stk} \rangle & \triangleq \text{stk} & \langle A \, \tau \rangle & \triangleq \langle A \rangle \\
\langle f \rangle & \triangleq f & \langle \alpha \rightarrow_{\kappa} A \rangle & \triangleq \langle A \rangle \\
\langle X \rangle & \triangleq X & \langle [\alpha \ll_{\kappa} \tau] B \rangle & \triangleq \langle B \rangle \\
\langle A \, B \rangle & \triangleq \langle A \rangle \, \langle B \rangle & \langle P \rightarrow_{\Delta} A \rangle & \triangleq P \rightarrow \langle A \rangle & P \neq \alpha \\
\langle A ; B \rangle & \triangleq \langle A \rangle ; \langle B \rangle & \langle [P \ll_{\Delta} A] B \rangle & \triangleq [P \ll \langle A \rangle ] \langle B \rangle & P \neq \alpha
\end{align*}
\]

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 (α →_{\kappa} 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.
Well-formed contexts

\[ \emptyset \vdash U \text{ ok} \]
\[ \Gamma \vdash \alpha \notin \text{Dom}(\Gamma) \]
\[ \Gamma, \alpha \vdash U \text{ ok} \]

Well-kindled Types

\[ \Gamma_1, \tau_1 : \Gamma_2 \vdash U \text{ ok} \]
\[ \Gamma_1, \alpha \vdash \tau_1 : \Gamma_2 \vdash \text{ ok} \]
\[ \Gamma \vdash \forall \alpha.\tau : \text{ ok} \]
\[ \Gamma \vdash \tau_1 : \tau_2 : \text{ ok} \]

Typing Rules (Figures 8 and 9).

A primitive polymorphic type assignment system was sketched in [CKL02] (without any metatheory). It proves judgment of the shape:

\[ \Gamma \vdash U \text{ ok} \quad \Gamma \vdash \tau : \text{ ok} \quad \Gamma \vdash P : \tau \quad \Gamma \vdash U : \tau \]

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

- (Term.Abs⁻): The domain of \( \Delta \) is given by the free-variables of \( P \), i.e. \( \text{Dom}(\Delta) = \text{Fv}(P) \).

- (Term.Abs⁺): This rule is not syntax directed; the classical side-condition about the freshness of \( \alpha \) is enforced by the well-formedness of the context in the premises.

- (Term.Appl⁻): This rule is not syntax directed; the type \( \tau_2 \) is guessed.

- (Term.Match⁻): The context \( \Delta \) is built from the free-variables of \( P \), i.e. \( \text{Dom}(\Delta) = \text{Fv}(P) \).
Well-formed Terms and Patterns

$$\Gamma \vdash_U \tau : * \quad \tau \neq \bot \quad \text{(Term-Stuck)}$$

$$\Gamma_1, X : \tau, \Gamma_2 \vdash_U ok \quad \text{(Term-Var)}$$

$$\Gamma_1, X : \tau, \Gamma_2 \vdash_U X : \tau \quad \text{(Term-Const)}$$

$$\Gamma \vdash_U U \tau : \tau \quad \text{(Term-App)}$$

$$\Gamma_1, \Delta \vdash_U P : \tau_1 \quad \text{‡}$$

$$\Gamma, \Delta \vdash_U U : \tau_2 \quad \text{Γ, Γ} \vdash_U \tau_1 \rightarrow \tau_2 : * \quad \text{(Term-Abs*)}$$

$$\Gamma \vdash_U U : \forall \alpha.\tau \quad \Gamma \vdash_U U : \forall \alpha.\tau \quad \text{Γ} \vdash_U \tau_1 \{\tau_2/\alpha\} \quad \text{(Term-App*)}$$

$$\Gamma \vdash_U U : \tau \quad \text{Γ} \vdash_U \tau_1 \quad \text{Γ} \vdash_U \tau_2 \quad \text{Γ} \vdash_U V : \tau_1 \quad \text{‡}$$

$$\Gamma \vdash_U U ; V : \tau \quad \text{(Term-Struct)}$$

$$\Gamma \vdash_U [P \llg V]U : \tau_2 \quad \text{(Term-Match*)}$$

where ‡ is [Dom(Δ) = Fv(P)]

Figure 9. The Type Assignment System for uRhoF

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 \( \Gamma \vdash_U U : \tau \) and \( U \mapsto \text{stk} \rho \sigma \delta V \), then \( \Gamma \vdash_U V : \tau \).

Proof By an induction on the derivation of \( \Gamma \vdash_U U : \tau \). □

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 \text{stk} \notin U, the following problem is undecidable:

- Type Inference: given \( \Gamma \) (gives meaning to constants), is there a type \( \tau \) such that \( \Gamma \vdash_U U : \tau \)?

Proof It follows a fortiori from the well known result of Wells [Wel99]. □
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 more dynamically typed) and intriguing for Maude, because the latter has a sophisticated form of polytypic programming [CDMO00,DM00], 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 [DM82] 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 $\llparenthesis \cdot \rrparenthesis$, at least for RhoF and uRhoF, induces an isomorphism between the derivations in the corresponding systems. More precisely, if $\text{Der}$ is a derivation in a typed system, by applying $\llparenthesis \cdot \rrparenthesis$ to every object (i.e. term, constructor, or kind) in $\text{Der}$, a valid derivation in the corresponding type assignment system is obtained. The other way around, every type assignment derivation can be obtained by applying $\llparenthesis \cdot \rrparenthesis$ to a typed one.

Definition 5.1 [Isomorphism]

Let $\text{Set}(\text{Der}_T)$ and $\text{Set}(\text{Der}_U)$ be the sets of all derivations in RhoF and uRhoF. Systems RhoF and uRhoF are isomorphic, via $\llparenthesis \cdot \rrparenthesis$, if and only if there are $\mathcal{F} : \text{Set}(\text{Der}_T) \Rightarrow \text{Set}(\text{Der}_U)$ and $\mathcal{G} : \text{Set}(\text{Der}_U) \Rightarrow \text{Set}(\text{Der}_T)$, such that:

(i) (Soundness) If $\text{Der}_T : \Gamma \vdash_I A : \tau$, then $\mathcal{F}(\text{Der}_T) : \llparenthesis \Gamma \rrparenthesis \vdash_U \llparenthesis A \rrparenthesis : \tau$.

(ii) (Completeness) If $\text{Der}_U : \Gamma' \vdash_U U : \tau$, then $\mathcal{G}(\text{Der}_U) : \Gamma \vdash_I A : \tau$, and $\llparenthesis \Gamma \rrparenthesis = \Gamma'$, with $\llparenthesis M \rrparenthesis = U$.

(iii) $\mathcal{F} \circ \mathcal{G}$ and $\mathcal{G} \circ \mathcal{F}$ are the identity on $\text{Set}(\text{Der}_U)$ and $\text{Set}(\text{Der}_T)$, respectively.
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 imply 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 [Liq96,Miq01].

**Theorem 5.2 (RhoF vs. uRhoF)**

Systems $\text{RhoF}$, and $\text{uRhoF}$ are isomorphic.

**Proof** The proof follows the same lines of [GHR93].

The relation with logic through the so-called Curry-Howard isomorphism [How80], or “formulae-as-types and proofs-as-terms” principle, has been deeply studied for the Lambda-calculus. As demonstrated in [CKL01b,BCKL03], 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 $\tau$ in the typed system $\text{RhoF}$, with a derivation $\text{Der}_T$, then the term $A$ can be seen as the coding of a logical proof, proving the formula $\varphi$ that can be interpreted as the type $\tau$ assigned to $A$.

For the typed system $\text{RhoF}$, 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 ‘formulae-as-types’ principle of Curry and Howard [How80] can be extended to the above type assignment systems as follows [Liq96]:

Given an untyped term $U$, if we can assign a type $\tau$ in the type assignment system $uRhoF$, with a derivation $\text{Der}_U$, then:

- $\text{Der}_U$ can be interpreted as the coding of a proof for the logic formulas $\varphi$ which corresponds to the interpretation of the type $\tau$ 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 $\varphi_i$’s that can be interpreted as the types $\tau_i$’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 “spoo” consistency at the price of only one algebraic constant $f$ declared in the context.

**Example 5.3 [Logical Inconsistency]**

Let $\Gamma \triangleq f : \bot \rightarrow \bot \rightarrow \bot$.

\[
\begin{align*}
\Gamma, X : \bot & \vdash X : \bot \\
\Gamma, X : \bot & \vdash f : \bot \rightarrow \bot \rightarrow \bot \\
\Gamma, X : \bot & \vdash f(X) : \bot \rightarrow \bot \\
\Gamma & \vdash f(X) \rightarrow_{(\bot : \bot)} X : (\bot \rightarrow \bot) \rightarrow \bot \\
\Gamma & \vdash X : \bot \rightarrow \bot \\
\Gamma & \vdash f(X) \rightarrow_{(\bot : \bot)} X : (X \rightarrow \bot) : \bot
\end{align*}
\]

The key point of this spoofer seems to be the contravariant position of $\bot$ in the typing of $f$: this allows to inhabit the 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 by Mendler [Men87] 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 \((Type\cdotArrow)\) as follows:

\[
\Gamma \vdash \tau_1 : \ast \quad \Gamma \vdash \tau_2 : \ast \quad \text{if } \tau_2 \equiv \tau_2^1 \rightarrow \ldots \tau_2^n \rightarrow \alpha \text{ then } \alpha > 0 \text{ in } \tau_1
\]

\[
\Gamma \vdash \tau_1 \rightarrow \tau_2 : \ast
\]

where \(\alpha > 0\) in \(\tau_1\) if:

(i) \(\alpha\) does not occur in \(\tau_1\);

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

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 \((Type\cdotArrow')\)) and every closed \(A\) and \(U\), and for a suitable \(\Gamma\) giving meaning to the algebraic constants of \(A\) and \(U\), respectively, the following holds: \(\Gamma \not\vdash T : \bot\), and \(\Gamma \not\vdash U : \bot\).

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 the Meseguer’s Conditional Rewriting Logic [Mes92].

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 expressivity. From the point of view of type inference, the main motivation, in introducing uRhoF, is 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 \textit{uRhoF} (called \textit{uRhoF}_{\text{mlet}}) featuring a restricted form of polymorphism à la Damas-Milner-Tofte and to customize of the well-known algorithm \textit{W} of Damas-Milner [DM82], conjecture (ii) of Exercise pp. 14 in [CKL02].

References


