Higher-Order Rewriting: Framework, Confluence and Termination

Jean-Pierre Jouannaud*

LIX/CNRS UMR 7161 & École Polytechnique, F-91400 Palaiseau
http://www.lix.polytechnique.fr/Labo/Jean-Pierre.Jouannaud/

1 Introduction

Equations are ubiquitous in mathematics and in computer science as well. This first sentence of a survey on first-order rewriting borrowed again and again characterizes best the fundamental reason why rewriting, as a technology for processing equations, is so important in our discipline [10]. Here, we consider *higher-order rewriting*, that is, rewriting higher-order functional expressions at higher-types. Higher-order rewriting is a useful generalization of first-order rewriting: by rewriting higher-order functional expressions, one can process abstract syntax as done for example in program verification with the prover Isabelle [27]; by rewriting expressions at higher-types, one can implement complex recursion schemas in proof assistants like Coq [12].

In our view, the role of higher-order rewriting is to design a type-theoretic frameworks in which computation and deduction are integrated by means of higher-order rewrite rules, while preserving decidability of typing and coherence of the underlying logic. The latter itself reduces to type preservation, confluence and strong normalization.

It is important to understand why there have been very different proposals for higher-order rewriting, starting with Klop's *Combinatory reduction systems* in 1980, Nipkow's *higher-order rewriting* in 1991 and Jouannaud and Okada's *executable higher-order algebraic specifications* in 1991 as well: these three approaches tackle the same problem, in different contexts, with different goals requiring different assumptions.

Jan Willem Klop was mostly interested in generalizing the theory of lambda calculus, and more precisely the confluence and finite developments theorems. Klop does not assume any type structure. As a consequence, the most primitive operation of rewriting, searching for a redex, is already a problem. Because he wanted to encode pure lambda calculus

* Project LogiCal, Pôle Commun de Recherche en Informatique du Plateau de Saclay, CNRS, École Polytechnique, INRIA, Université Paris-Sud.

and other calculi as combinatory reduction systems, he could not stick to a pure syntactic search based on first-order pattern matching. He therefore chose to search via finite developments, the only way to base a finite search on beta-reduction in the absence of typing assumptions. And because of his interest in simulating pure lambda calculi, he had no real need for termination, hence concentrated on confluence results. Therefore, his theory in its various incarnations is strongly influenced by the theory of residuals initially developed for the pure lambda calculus.

Nipkow was mainly interested in investigating the meta-theory of Isabelle and in proving properties of functional programs by rewriting, goals which are actually rather close to the previous one. Functional programs are typed lambda terms, hence he needed a typed structure. He chose searching via higher-order pattern matching, because plain pattern matching is too poor for expressing interesting transformations over programs with finitely many rules. This choice of higher-order pattern matching instead of finite developments is of course very natural in a typed framework. He assumes termination for which proof methods were still lacking at that time. His (local) confluence results rely on the computation of higher-order critical pairs -because higher-order pattern matching is used for searching redexes- via higher-order unification. Nipkow restricted lefthand sides of rewrite rules to be patterns in the sense of Miller [25]. The main reason for this restriction is that higher-order pattern matching and unification are tractable in this case which, by chance, fits well with most intended applications.

Jouannaud and Okada were aiming at developping a theory of typed rewrite rules that would generalize the notion of recursor in the calculus of inductive constructions, itself generalizing Gödel's system T. This explains their use of plain pattern matching for searching a redex: there is no reason for a more sophisticated search with recursors. In this context, the need for strongly terminating calculi has two origins: ensuring consistency of the underlying logic, that is, the absence of a proof for falsity on the one hand, and decidability of the type system in the presence of dependent types on the other hand. Confluence is needed as well, of course, as is type preservation. The latter is easy to ensure while the former is based on the computation of first-order critical pairs -because first-order pattern matching is used for searching redexes. This explains the emphasis on termination criteria in this work and its subsequent developments.

Our goal in this paper is to present a unified framework borrowed from Jouannaud, Rubio and van Raamsdonk [22] for the most part, in which redexes can be searched for by using either plain or higher-order

rewriting, confluence can be proved by computing plain or higher-order critical pairs, and termination can be proved by using the higher-order recursive path ordering of Jouannaud and Rubio [19].

We first present examples showing the need for both search mechanisms based on plain and higher-order pattern matching on the one hand, and for a rich type structure on the other hand. These examples show the need for rules of higher type, therefore contradicting a common belief that application makes rules of higher type unnecessary. They also recall that Klop's idea of variables with arities is very handy. Then, we present our framework in more detail, before we address confluence issues, and finally termination criteria. Missing notations and terminology used in rewriting or type theory can be found in [10, 2].

2 Examples

We present here by means of examples the essential features of the three aforementionned approaches to higher-order rewriting. Rather than comparing their respective expressivity [32], we use our unified framework to show why they are important and how they can be smoothly integrated. Framework and syntax are explained in detail in Section 3, but the necessary information is already provided here to make this section self-contained to anybody who is familiar with typed lambda calculi.

Language. It is our assumption that our examples extend a typed lambda calculus, in which abstraction and application (always written explicitely) are written $\lambda x : \sigma.u$ and $@(u, v)$ respectively where u, v are terms, x is a variable and σ a type. We sometimes drop types in abstractions, and write $\lambda xy.u$ for $\lambda x.(\lambda y.u)$, assuming that the scope of an abstraction extends as far to the right as possible. A variable not in the scope of an abstraction is said to be *free*, otherwise it is *bound*. An expression is *ground* if it has no free variable. The (right-associative) type constructor $\rightarrow$ for functional types is our main type constructor, apart from user-defined ones. We will also need a weak notion of polymorphism, requiring the use of a special (untypable) constant $*$ denoting an arbitrary type. We do not have product types, unless introduced by the user. Recall that lambda calculus is a formal model in which functional computations are described by three (higher-order) rewrite rules, beta-reduction, eta-reduction and alpha-conversion:

beta	$@(\lambda x.u, v) \longrightarrow_\beta u\{x \mapsto v\}$	
eta	$\lambda x.@(u, x) \longrightarrow_\eta u$	where x is not a free variable of u
alpha	$\lambda y.u \longrightarrow_\alpha \lambda x.u\{y \mapsto x\}$	where x is not a free variable of u

In these rules, u, v stand for arbitrary terms, making them rule schemas rather than true rewrite rules. The notation $u\{x \mapsto v\}$ stands for substitution of x by v in u. *Variable capture*, that is, a free variable of v becoming bound after substitution, is disallowed which may force renaming bound variables via alpha-conversion before instantiation can take place. The second rule can also be used from right-to-left, in which case it is called an *expansion*. These rules define an equivalence over terms, the *higher-order equality* $=_{\beta\eta}$, which can be decided by computing normal forms: by using beta and eta both as reductions yielding the beta-eta-normal form; or by using beta as a reduction and eta as an expansion (for terms which are not the left argument of an application, see Section 3 for details) yielding the eta-long beta-normal form. Given two terms u and v, *first-order pattern matching* (resp. *unification*) computes a subtitution σ such that $u = v\sigma$ (resp. $u\sigma = v\sigma$). *Plain* and *syntactic* are also used instead of first-order, for qualifying pattern matching, unification or rewriting. *Higher-order pattern matching* (resp. *unification*) computes a subtitution σ such that $u =_{\beta\eta} v\sigma$ (resp. $u\sigma =_{\beta\eta} v\sigma$). Of course, such substitutions may not exist. First-order pattern matching and unification are decidable in linear time. Higher-order unification is undecidable, while the exact status of higher-order matching is unknown at orders 5 and up [11].

Our examples come in two parts, a signature for the constants and variables, and a set of higher-order rewrite rules added to the rules of the underlying typed lambda calculus. We use a syntax *à la OBJ*, with keywords introducing successively type constants (`Typ`), type variables (`Tva`), term variables (`Var`), constructors (`Con`), defined function symbols (`Ope`), and rewrite rules (`Prr` and `Hor`). Defined function symbols occur as head operators in lefthand sides of rules, while constructors may not. With some exceptions, we use small letters for constants, greek letters for type variables, capital latin letters for free term variables, and small latin letters for bound term variables. Due to the hierarchichal structure of our specifications, and the fact that a type declaration binds whatever comes after, the polymorphism generated by the type variables is weak, as in Isabelle: there is no need for explicit quantifiers which are all external.

Finally, our framework is typed with arities: besides having a type, constants also have an arity which is indicated by writing a double arrow $\Rightarrow$ instead of a single arrow $\rightarrow$ to separate input types from the output type in the typing declarations. The double arrow does not appear when there are no input types. We write $f(u_1, \ldots, u_n)$ when f has arity $n > 0$ and simply f when $n = 0$. In general, the use of arities facilitates the reading. Just like constants, variables also will have arities.

Gödel's system T. We give a polymorphic version of Gödel's system T, a simply typed lambda calculus in which natural numbers are represented in Peano notation. This example has an historical significance: it is the very first higher-order rewrite system *added* to a typed lambda calculus, introduced by Gödel to study the logic of (a fragment of) arithmetic; it plaid a fundamental role in the understanding of the Curry-Howard isomorphism which led to the definition of System F by Girard.

Example 1. **Recursor for natural numbers**

$$
\begin{array}{lrcl}
\texttt{Tva} & \alpha & : & * \\
\texttt{Typ} & \mathbb{N} & : & * \\
\texttt{Con} & \mathsf{0} & : & \mathbb{N} \\
\texttt{Con} & \mathsf{s} & : & \mathbb{N} \Rightarrow \mathbb{N} \\
\texttt{Ope} & \mathsf{rec} & : & \mathbb{N} \to \alpha \to (\mathbb{N} \to \alpha \to \alpha) \Rightarrow \alpha \\
\texttt{Var} & \mathsf{X} & : & \mathbb{N} \\
\texttt{Var} & \mathsf{U} & : & \alpha \\
\texttt{Var} & \mathsf{Y} & : & \mathbb{N} \to \alpha \to \alpha
\end{array}
$$

$$
\begin{array}{ll}
\texttt{Prr} & \mathsf{rec}(\mathsf{0},\mathsf{U},\mathsf{Y}) \to \mathsf{U} \\
\texttt{Prr} & \mathsf{rec}(\mathsf{s}(\mathsf{X}),\mathsf{U},\mathsf{Y}) \to @(\mathsf{Y},\mathsf{X},\mathsf{rec}(\mathsf{X},\mathsf{U},\mathsf{Y}))
\end{array}
$$

In this example $\mathbb{N}$ is the only type constant and α the only type variable. The constants $\mathsf{0}$ and s are the two constructors for the type $\mathbb{N}$ of natural numbers, as it can be observed from their output type. All variables have arity zero. The rec operator provides us with *higher-order primitive recursion*. It can be used to define new functions, such as addition, multiplication, exponentiation or even the Ackermann function by choosing appropriate instantiations for the higher-order variables U and X in the rules given in Example 1. For example,

$$
\begin{array}{lll}
\texttt{Var} & \mathsf{M},\mathsf{N} : \mathbb{N} & \\
\texttt{Prr} & \mathsf{plus}(\mathsf{N},\mathsf{M}) & \to \mathsf{rec}(\mathsf{N},\mathsf{M},\lambda \mathsf{z}_1\mathsf{z}_2.\mathsf{s}(\mathsf{z}_2)) \\
\texttt{Prr} & \mathsf{mul}(\mathsf{N},\mathsf{M}) & \to \mathsf{rec}(\mathsf{N},\mathsf{0},\lambda \mathsf{z}_1\mathsf{z}_2.\mathsf{plus}(\mathsf{M},\mathsf{z}_2))
\end{array}
$$

The precise understanding of the recursor requires some familiarity with the so-called Curry-Howard isomorphism, in which types are propositions in some fragment of intuitionistic logic (here, a quantified propositionnal fragment), terms of a given type are proofs of the corresponding proposition, and higher-order rules describe proof transformations. In system T, rec can be intuitively interpreted as carrying the proof of a proposition α (the output type of rec) done by induction over the natural numbers. Assuming that α has the form $\forall n.P(n)$ (beware that this

proposition is not a type here, we would need a type system à la Coq for that), the variable U is then a proof of $P(\mathsf{0})$, while Y is a function which takes a natural number n and a proof of $P(n)$ as inputs and yields a proof of $P(\mathsf{s}(n))$ as output. It is now easy to see that the first rule equates two proofs of $P(\mathsf{0})$. Further, since $\mathsf{rec}(\mathsf{X},\mathsf{U},\mathsf{Y})$ in the rigthand side of the second rule is a proof of $P(\mathsf{X})$, that rule equates two proofs of $P(\mathsf{s}(\mathsf{X}))$.

This simple example is already quite interesting in our view.

First, it is based on the use of plain pattern matching. This is always so with recursors for inductive types and is indicated here by using the keyword `Prr`. This comes from the fact that a ground expression of an inductive type (like $\mathbb{N}$) which is in normal form must be headed by a constructor ($\mathsf{0}$ or s for $\mathbb{N}$). Now, pattern matching an expression in normal form (like $\mathsf{0}$ or $\mathsf{s}(u)$ for some normal form u) with respect to the terms 0 and $\mathsf{s}(\mathsf{X})$ (in the case of rec) or with respect to the variable N (in the case of $\mathsf{plus},\mathsf{mul}$) does not need higher-order pattern matching since beta- and eta-reductions can only occur inside variables (X or N).

Second, there is no way to define recursors by *a finite number of higher-order rules* in the absence of polymorphism. A description saying that U is a variable of an arbitrary ground type amounts to have one rule for each ground type, which does not fit our purpose: to give a finite specification for system T.

Finally, observe that rewriting expressions for which subexpressions of type $\mathbb{N}$ are ground results in a normal form in which the recursor does not appear anymore. In the OBJ jargon, the operator `rec` is *sufficiently defined*. The fact that all defined operators are sufficiently defined is crucial for encoding recursion by `rec`.

Polymorphic lists.

Example 2. **Recursors for lists**

`Typ`	list	:	$* \Rightarrow *$
`Tva`	α, β	:	$*$
`Con`	nil	:	$\mathsf{list}(\alpha)$
`Con`	cons	:	$\alpha \to \mathsf{list}(\alpha) \Rightarrow \mathsf{list}(\alpha)$
`Ope`	map	:	$\mathsf{list}(\alpha) \to (\alpha \to \beta) \Rightarrow \mathsf{list}(\beta)$
`Var`	H	:	α
`Var`	T	:	$\mathsf{list}(\alpha)$
`Var`	F	:	$\alpha \to \beta$
`Prr`	$\mathsf{map}(\mathsf{nil},\mathsf{F})$	$\to$	nil
`Prr`	$\mathsf{map}(\mathsf{cons}(\mathsf{H},\mathsf{T}),\mathsf{F})$	$\to$	$\mathsf{cons}(@(\mathsf{F},\mathsf{H}),\mathsf{map}(\mathsf{T},\mathsf{F}))$

This example familiar to Lisp programmers shows that the above final remark applies to any inductive type, such as polymorphic lists. Here, list is a type operator of arity one, therefore taking an arbitrary type as input. map returns the list of applications of the function F given as second argument to the elements of the list given as first argument.

Example 3 extends Example 2 with a parametric version of insertion sort (called sort), which takes a list as input, sorts the tail and then inserts the head at the right place by using a function insert, whose second argument is therefore a sorted list. The additional parameters X, Y in both sort and insert stand for functions selecting one of their two arguments with respect to some ordering. Instantiating them yields particular sorting algorithms, as for example ascending − sort. The additional signature declarations are omitted, as well as the keyword `Prr`.

Example 3. **Parametric insertion sort**

$$\begin{aligned}
\mathsf{max}(0,\mathsf{X}) &\to \mathsf{X} & \mathsf{max}(\mathsf{X},0) &\to \mathsf{X}\\
\mathsf{max}(\mathsf{s}(\mathsf{X}),\mathsf{s}(\mathsf{Y})) &\to \mathsf{s}(\mathsf{max}(\mathsf{X},\mathsf{Y}))\\
\mathsf{min}(0,\mathsf{X}) &\to 0 & \mathsf{min}(\mathsf{X},0) &\to 0\\
\mathsf{min}(\mathsf{s}(\mathsf{X}),\mathsf{s}(\mathsf{Y})) &\to \mathsf{s}(\mathsf{min}(\mathsf{X},\mathsf{Y}))\\
\mathsf{insert}(\mathsf{N},\mathsf{nil},\mathsf{X},\mathsf{Y}) &\to \mathsf{cons}(\mathsf{N},\mathsf{nil})\\
\mathsf{insert}(\mathsf{N},\mathsf{cons}(\mathsf{M},\mathsf{T}),\mathsf{X},\mathsf{Y}) &\to \mathsf{cons}(@(\mathsf{X},\mathsf{N},\mathsf{M}),\mathsf{insert}(@(\mathsf{Y},\mathsf{N},\mathsf{M}),\mathsf{T},\mathsf{X},\mathsf{Y}))\\
\mathsf{sort}(\mathsf{nil},\mathsf{X},\mathsf{Y}) &\to \mathsf{nil}\\
\mathsf{sort}(\mathsf{cons}(\mathsf{N},\mathsf{T}),\mathsf{X},\mathsf{Y}) &\to \mathsf{insert}(\mathsf{N},\mathsf{sort}(\mathsf{T},\mathsf{X},\mathsf{Y}),\mathsf{X},\mathsf{Y})\\
\mathsf{ascending}-\mathsf{sort}(\mathsf{L}) &\to \mathsf{sort}(\mathsf{L},\lambda\mathsf{xy}.\mathsf{min}(\mathsf{x},\mathsf{y}),\lambda\mathsf{xy}.\mathsf{max}(\mathsf{x},\mathsf{y}))\\
\mathsf{descending}-\mathsf{sort}(\mathsf{L}) &\to \mathsf{sort}(\mathsf{L},\lambda\mathsf{xy}.\mathsf{max}(\mathsf{x},\mathsf{y}),\lambda\mathsf{xy}.\mathsf{min}(\mathsf{x},\mathsf{y}))
\end{aligned}$$

As this example shows, many programs can be defined by first-order pattern matching, a well-known fact exploited by most modern functional programming languages, such as those of the ML family. Again, we could (and should) prove that these new defined operators are sufficiently defined, but the argument is actually the same as before.

Differentiation. We now move to a series of examples showing the need for higher-order pattern matching, which will be indicated by using the second keyword `Hor` for rules. First, we need some more explanations about the typed lambda calculus. $\mathbb{N}$ and $\mathsf{list}(\mathbb{N})$ are called *data types*, while $\mathbb{N} \to \mathbb{N}$, which is headed by $\to$, is a *functional type*. Constant data types like $\mathbb{N}$ are also called *basic types*. In Nipkow's work, the lefthand and righthand side of higher-order rules must have the same basic type, a condition which is usually enforced when needed by applying a term of a functional type to a sequence of variables of the appropriate types. All

terms must be in eta-long beta-normal form, forcing us to write $\lambda \mathsf{x}.@(\mathsf{F},\mathsf{x})$ instead of simply F. Lefthand sides of rules must be *patterns* [25], an assumption frequently met in practice which makes both higher-order pattern matching and unification decidable in linear time. Finally, there is no notion of constructor and defined symbol, and no recursors coming along with. We choose arbitrarily to have all symbols as defined. We give only one rule, the others should be easily guessed by the reader.

Example 4. **Differentiation 1**

$$
\begin{array}{lrcl}
\texttt{Typ} & \mathbb{R} & : & * \\
\texttt{Ope} & \mathsf{sin},\mathsf{cos} & : & \mathbb{R} \to \mathbb{R} \\
\texttt{Ope} & \mathsf{mul} & : & \mathbb{R} \to \mathbb{R} \to \mathbb{R} \\
\texttt{Ope} & \mathsf{diff} & : & (\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R} \\
\texttt{Var} & \mathsf{y} & : & \mathbb{R} \\
\texttt{Var} & \mathsf{F} & : & \mathbb{R} \to \mathbb{R}
\end{array}
$$

$$
\begin{array}{l}
\texttt{Hor}\ @(\mathsf{diff}, \lambda \mathsf{x} : \mathbb{R}.\, @(\mathsf{sin}, @(\mathsf{F},\mathsf{x})),\mathsf{y}) \to \\
\qquad @(\mathsf{mul}, @(\mathsf{cos}, @(\mathsf{F},\mathsf{y})), @(\mathsf{diff}, \lambda \mathsf{x} : \mathbb{R}.@(\mathsf{F},\mathsf{x}),\mathsf{y}))
\end{array}
$$

Note that all function symbols and variables have arity zero, since there is no arity in our sense in Nipkow's framework. Both sides of the rule have type $\mathbb{R}$: diff computes the differential of its first argument at the point given as second argument. Unlike the previous examples, these rules use higher-order pattern matching, and this is necessary here to compute the derivative of the function sin itself. Clearly, $@(\mathsf{diff}, \lambda \mathsf{x}.@(\mathsf{sin},\mathsf{x}))$ does not match the lefthand side of rule. Let us instantiate the variable F of the lefthand side of rule by the identity function $\lambda x.x$, resulting in the expression $@(\mathsf{diff}, \lambda \mathsf{x}.\, @(\mathsf{sin}, @(\lambda \mathsf{x}.\mathsf{x},\mathsf{x})),\mathsf{y})$ which beta-reduces to the expected result $@(\mathsf{diff}, @(\mathsf{sin},\mathsf{x})),\mathsf{y})$. This shows that the latter expression higher-order matches the lefthand side of rule. Using plain pattern matching would require infinitely many rules, one for each possible instantiation of F requiring a beta-reduction. Incorporating beta-reduction into the matching process allows one to have a single rule for all cases.

Using rules of higher-order type as well as function symbols and variables of non-zero arity is possible thanks to a generalisation of Nipkow's work [22], resulting in the following new version of the same example:

Example 5. **Differentiation 2**

$$
\begin{array}{lrcl}
\texttt{Typ} & \mathbb{R} & : & * \\
\texttt{Ope} & \mathsf{sin}, \mathsf{cos} & : & \mathbb{R} \Rightarrow \mathbb{R} \\
\texttt{Ope} & \mathsf{mul} & : & (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R}) \Rightarrow (\mathbb{R} \to \mathbb{R}) \\
\texttt{Ope} & \mathsf{diff} & : & (\mathbb{R} \to \mathbb{R}) \Rightarrow (\mathbb{R} \to \mathbb{R}) \\
\texttt{Var} & \mathsf{F} & : & \mathbb{R} \Rightarrow \mathbb{R} \\
\texttt{Hor} & \mathsf{diff}(\lambda \mathsf{x} : \mathbb{R}.\, \mathsf{sin}(\mathsf{F}(\mathsf{x})) \to \mathsf{mul}(\lambda \mathsf{x} : \mathbb{R}.\, \mathsf{cos}(\mathsf{F}(\mathsf{x})), \mathsf{diff}(\lambda \mathsf{x} : \mathbb{R}.\mathsf{F}(\mathsf{x}))) & &
\end{array}
$$

Here, the rule has the higher-order type $\mathbb{R} \Rightarrow \mathbb{R}$, making diff the true differential of the function given as argument. In this format, eta-expansion is controlled by the systematic use of arities. This use of arities for replacing applications by parentheses should be considered here as a matter of style: one can argue that arities are not really needed for that purpose, since, traditionally, the application operator is not explicitly written in the lambda calculus: some write $(M\ N)$ for $@(M, N)$ while others favour $M(N)$ instead. But this is a convention. In both cases, the conventional expression is transformed by the parser into the one with explicit application. Here, the syntax forces us to write $@(M, N)$ when M is not a variable or is a variable with arity zero, and $M(N)$ when M is a variable with arity 1. More convincing advantages are discussed later.

To get the best of our format, we can declare F as a function symbol of arity zero and use the beta-eta-normal form instead of the eta-long beta-normal form. With this last change, the rule becomes:

$$
\begin{array}{lrcl}
\texttt{Var} & \mathsf{F} & : & \mathbb{R} \to \mathbb{R} \\
\texttt{Hor} & \mathsf{diff}(\mathsf{sin}(\mathsf{F})) \to \mathsf{mul}(\mathsf{cos}(\mathsf{F}), \mathsf{diff}(\mathsf{F})) & &
\end{array}
$$

Simply typed lambda calculus. We end up this list with Klop's favorite example, showing the need for arities of variables in order to encode the lambda calculus. The challenge here is to have true rewrite rules for beta-reduction and eta-reduction: the usual side condition for the eta-rule must be eliminated. This is made possible by allowing us to control which variables can or cannot be captured through substitution when replacing a variable with arity: a substitute for a variable of arity n must be an abstraction of n different bound variables. For example, a variable of arity one depends upon a single variable, like X which must be written $X(x)$ for some variable x bound above, and replaced by an abstraction $\lambda x.u$ for some term u. The instance of $X(x)$ will then become $@(\lambda x.u, x)$ and beta-reduce to u. This idea due to Klop is indeed very much related to Miller's notion of pattern: in a pattern, every occurence of X must be of

the form $X(x)$ with x bound above if X has arity one, and it becomes then natural to define u as the substitute for $X(x)$ rather than $\lambda x.u$ as a substitute for X which does not exist as a syntactic term. We will later see that this relationship is stronger than anticipated.

Example 6. **Simply typed lambda calculus**

$$\begin{array}{lrcl}
\texttt{Typ} & \alpha, \beta & : & * \\
\texttt{Ope} & \mathsf{app} & : & (\alpha \to \beta) \to \alpha \Rightarrow \beta \\
\texttt{Ope} & \mathsf{abs} & : & (\alpha \to \beta) \Rightarrow (\alpha \to \beta) \\
\texttt{Var} & \mathsf{U} & : & \alpha \Rightarrow \beta \\
\texttt{Var} & \mathsf{V} & : & \alpha \\
\texttt{Var} & \mathsf{X} & : & \alpha \to \beta
\end{array}$$

$$\begin{array}{ll}
\texttt{Hor} & \mathsf{app}(\mathsf{abs}(\lambda \mathsf{x} : \alpha.\mathsf{U}(\mathsf{x})), \mathsf{V}) \to \mathsf{U}(\mathsf{V}) \\
\texttt{Hor} & \mathsf{abs}(\lambda \mathsf{x} : \alpha.\mathsf{app}(\mathsf{X}, \mathsf{x})) \to \mathsf{X}
\end{array}$$

The beta-rule shows the use of a variable of arity one in order to internalize the notion of substitution, while the eta-rule shows the use of a variable of arity zero in order to eliminate the condition that x does not occur free in an instance of X: since X has arity zero, it cannot have an abstraction for substitute. And because variable capture is forbidden by the definition of substitution, x cannot occur in the substitute. The use of arity 1 for U is not essential here, since we could also choose the arity zero to the price of replacing the first rule by the variant

$$\texttt{Hor} \quad \mathsf{app}(\mathsf{abs}(\mathsf{U}), \mathsf{V}) \to @(\mathsf{U}, \mathsf{V}).$$

The example is a variation of Klop's, since we actually model a lambda calculus with simple types. It also shows the need of a rich enough type structure for specifying an example even as simple as this one.

3 Polymorphic Higher-Order Algebras

This section introduces the framework of polymorphic algebras [20, 22]. The use of polymorphic operators requires a rather heavy apparatus.

3.1 Types

Given a set $\mathcal{S}$ of *sort symbols* of a fixed arity, denoted by $s : *^n \Rightarrow *$, and a set $\mathcal{S}^\forall$ of *type variables*, the set $\mathcal{T}_{\mathcal{S}^\forall}$ of *polymorphic types* is generated from these sets by the constructor $\to$ for *functional types*:

$$\begin{array}{c}
\mathcal{T}_{\mathcal{S}^\forall} := \alpha \mid s(\mathcal{T}^n_{\mathcal{S}^\forall}) \mid (\mathcal{T}_{\mathcal{S}^\forall} \to \mathcal{T}_{\mathcal{S}^\forall}) \\
\text{for } \alpha \in \mathcal{S}^\forall \text{ and } s : *^n \Rightarrow * \ \in \mathcal{S}
\end{array}$$

where $s(\mathcal{T}^n_{\mathcal{S}^\forall})$ denotes an expression of the form $s(t_1, \ldots, t_n)$ with $t_i \in \mathcal{T}_{\mathcal{S}^\forall}$ for all $i \in [1..n]$. We use $\mathcal{V}ar(\sigma)$ for the set of (type) variables of the type $\sigma \in \mathcal{T}_{\mathcal{S}^\forall}$. When $\mathcal{V}ar(\sigma) \neq \emptyset$, the type σ is said to be *polymorphic* and *monomorphic* otherwise. A type σ is *functional* when headed by the $\rightarrow$ symbol, a *data type* when headed by a sort symbol (*basic* when the sort symbol is a constant). $\rightarrow$ associates to the right.

A *type substitution* is a mapping from $\mathcal{S}^\forall$ to $\mathcal{T}_{\mathcal{S}^\forall}$ extended to an endomorphism of $\mathcal{T}_{\mathcal{S}^\forall}$. We write $\sigma\xi$ for the application of the type substitution ξ to the type σ. We denote by $\mathcal{D}om(\sigma) = \{\alpha \in \mathcal{S}^\forall \mid \alpha\sigma \neq \alpha\}$ the domain of $\sigma \in \mathcal{T}_{\mathcal{S}^\forall}$, by $\sigma|_\mathcal{V}$ its restriction to the domain $\mathcal{D}om(\sigma) \cap \mathcal{V}$, by $\mathcal{R}an(\sigma) = \bigcup_{\alpha \in \mathcal{D}om(\sigma)} \mathcal{V}ar(\alpha\sigma)$ its *range*. By a renaming of the type σ apart from $V \subset \mathcal{X}$, we mean a type $\sigma\xi$ where ξ is a type renaming such that $\mathcal{D}om(\xi) = \mathcal{R}an(\sigma)$ and $\mathcal{R}an(\xi) \cap \mathcal{V} = \emptyset$.

We shall use α, β for type variables, $\sigma, \tau, \rho, \theta$ for arbitrary types, and ξ, ζ to denote type substitutions.

3.2 Signatures

We are given a set of function symbols denoted by the letters f, g, h, which are meant to be algebraic operators equiped with a fixed number n of arguments (called the *arity*) of respective types $\sigma_1 \in \mathcal{T}_{\mathcal{S}^\forall}, \ldots, \sigma_n \in \mathcal{T}_{\mathcal{S}^\forall}$, and an *output type* $\sigma \in \mathcal{T}_{\mathcal{S}^\forall}$ such that $\mathcal{V}ar(\sigma) \subseteq \bigcup_i \mathcal{V}ar(\sigma_i)$ if $n > 0$. We call *aritype* the expression $\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \Rightarrow \sigma$ when $n > 0$ and σ when $n = 0$, which can be seen as a notation for the pair made of a type and an arity. The condition $\mathcal{V}ar(\sigma) \subseteq \bigcup_i \mathcal{V}ar(\sigma_i)$ if $n > 0$ ensures that aritypes encode a logical proposition universally quantified outside, making our type system to come both rich and simple enough. Let $\mathcal{F}$ be the set of all function symbols:

$$\mathcal{F} = \biguplus_{\sigma_1, \ldots, \sigma_n, \sigma} \mathcal{F}_{\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \Rightarrow \sigma}$$

The membership of a given function symbol f to the set $\mathcal{F}_{\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \Rightarrow \sigma}$ is called a *type declaration* and written $f : \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \Rightarrow \sigma$. A type declaration is *first-order* if it uses only sorts, and higher-order otherwise. It is *polymorphic* if it uses some polymorphic type, otherwise, it is *monomorphic*. Polymorphic type declarations are implicitely universally quantified: they can be renamed arbitrarily. Note that type instantiation does not change the arity of a function symbol.

Alike function symbols, variables have an aritype. A variable declaration will therefore take the same form as a function declaration.

3.3 Terms

The set $\mathcal{T}(\mathcal{F}, \mathcal{X})$ of (raw) *terms* is generated from the signature $\mathcal{F}$ and a denumerable set $\mathcal{X}$ of arityped variables according to the grammar:

$$\mathcal{T} := (\lambda\mathcal{X} : \mathcal{T}_{\mathcal{S}^\forall}.\mathcal{T}) \mid @(\mathcal{T}, \mathcal{T}) \mid \mathcal{X}(\mathcal{T}, \ldots, \mathcal{T}) \mid \mathcal{F}(\mathcal{T}, \ldots, \mathcal{T}).$$

$\overline{s}$ will ambiguously denote a list, a set or a multiset of terms $s_1, \ldots, s_n$. Terms of the form $\lambda x : \sigma.u$ are called *abstractions*, the type σ being possibly omitted. Because a type is a particular aritype, bound variables have arity zero. $@(u, v)$ denotes the application of u to v. Parentheses are omitted for function or variable symbols of arity zero. The term $@(\overline{v})$ is called a (partial) *left-flattening* of $s = @((\ldots @(v_1, v_2)) \ldots v_n)$, v_1 being possibly an application itself. $\mathcal{V}ar(t)$ is the set of free term variables of t.

Terms are identified with finite labeled trees by considering $\lambda x : \sigma._-$, for each variable x and type σ, as a unary function symbol taking a term u as argument to construct the term $\lambda x : \sigma.u$. *Positions* are strings of positive integers. Λ and $\cdot$ denote respectively the empty string (root position) and string concatenation. $\mathcal{P}os(t)$ is the set of positions in t. $t|_p$ denotes the *subterm* of t at position p. Replacing $t|_p$ at position p in t by u is written $t[u]_p$. The notation $t[]_p$ stands for a *context* waiting for a term to fill its hole.

3.4 Typing rules

Definition 1. *An* environment Γ *is a finite set of pairs written as* $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$*, where* x_i *is a variable,* σ_i *is an aritype, and* $x_i \neq x_j$ *for* $i \neq j$*.* $\mathcal{V}ar(\Gamma) = \{x_1, \ldots, x_n\}$ *is the set of variables of* Γ*. The size* $|\Gamma|$ *of the environment* Γ *is the sum of the sizes of its constituents. Given two environments* Γ *and* Γ'*, their* composition *is the environment* $\Gamma \cdot \Gamma' = \Gamma' \cup \{x : \sigma \in \Gamma \mid x \notin \mathcal{V}ar(\Gamma')\}$*. Two environments* Γ *and* Γ' *are* compatible *if* $\Gamma \cdot \Gamma' = \Gamma \cup \Gamma'$*.*

Our typing judgments are written as $\Gamma \vdash_{\mathcal{F}} s : \sigma$. A term s has type σ in the environment Γ if and only if the judgment $\Gamma \vdash_{\mathcal{F}} s : \sigma$ is provable in the inference system of Figure 1. Given an environment Γ, a term s is *typable* if there exists a type σ such that $\Gamma \vdash_{\mathcal{F}} s : \sigma$.

Remember our convention that function and variable symbols having arity zero come without parentheses. When writing a jugement $\Gamma \vdash_{\mathcal{F}} s : \sigma$, we must make sure that σ is a type in the environment defined by the signature. Types are indeed unisorted first-order terms (of sort $*$). Since there is only one sort, and type symbols have a fixed arity, verifying

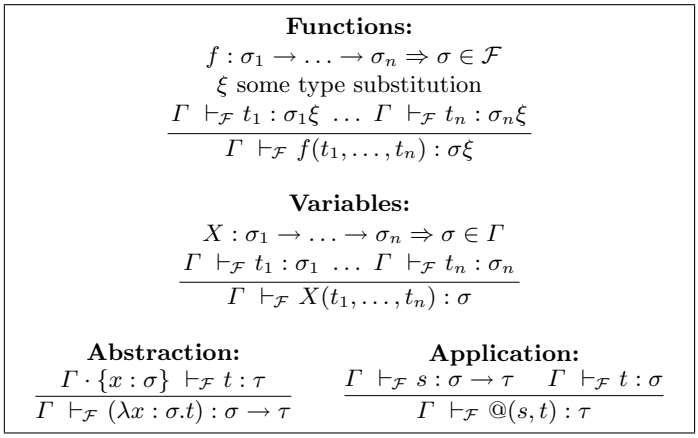

Fig. 1. The type system for polymorphic higher-order algebras with arities

that an expression is a type amounts to check that all symbols occuring in the expression are sort symbols or type variables, and that all sort symbols in the expression have the right number of types as inputs, an easily decidable property usually called *well-formedness*.

This typing system enjoys the *unique typing* property: given an environment Γ and a typable term u, it can be easily shown by induction on u that there exists a unique type σ such that $\Gamma \vdash_{\mathcal{F}} u : \sigma$.

Note that type substitutions apply to types in terms: $x\xi = x$, $(\lambda x : \sigma.s)\xi = \lambda x : \sigma\xi.s\xi$, $@(u, v)\xi = @(u\xi, v\xi)$, and $f(\overline{u})\xi = f(\overline{u}\xi)$.

3.5 Substitutions

Definition 2. *A (term)* substitution $\gamma = \{(x_1 : \sigma_1) \mapsto (\Gamma_1, t_1), \ldots, (x_n : \sigma_n) \mapsto (\Gamma_n, t_n)\}$*, is a finite set of quadruples made of a variable symbol, an aritype, an environment and a term, such that*

(i) Let $\sigma_i = \tau_{i_1} \to \ldots \to \tau_{i_p} \Rightarrow \tau_i$*. Then* $t_i = \lambda y_{i_1} : \tau_{i_1} \ldots y_{i_p} : \tau_{i_p}.u_i$ *for distinct variables* $y_{i_1}, \ldots, y_{i_p}$ *and term* $u_i \neq x_i$ *such that* $\Gamma_i \vdash_{\mathcal{F}} t_i : \sigma_i$*.*

(ii) $\forall i \neq j \in [1..n]$*,* $x_i \neq x_j$*, and*

(iii) $\forall i \neq j \in [1..n]$*,* Γ_i *and* Γ_j *are compatible environments.*

We may omit the aritype σ_i *and environment* Γ_i *in* $(x_i : \sigma_i) \mapsto (\Gamma_i, t_i)$*.*

The set of (input) variables of the substitution γ *is* $\mathcal{V}ar(\gamma) = \{x_1, \ldots, x_n\}$*, its* domain *is the environment* $\mathcal{D}om(\gamma) = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ *while its* range *is the environment (by assumption (iii))* $\mathcal{R}an(\gamma) = \bigcup_{i \in [1..n]} \Gamma_i$*.*

Definition 3. *A substitution* γ *is* compatible *with an environment* Γ *if*

(i) $\mathcal{D}om(\gamma)$ *is compatible with* Γ*,*
(ii) $\mathcal{R}an(\gamma)$ *is compatible with* $\Gamma \setminus \mathcal{D}om(\gamma)$*.*
We will also say that γ *is compatible with the judgement* $\Gamma \vdash_{\mathcal{F}} s : \sigma$*.*

Definition 4. *A substitution* γ *compatible with a judgement* $\Sigma \vdash_{\mathcal{F}} s : \sigma$ *operates as an endomorphism on* s *and yields the* instance $s\gamma$ *defined as:*

If $s = @(u, v)$	*then* $s\gamma = @(u\gamma, v\gamma)$
If $s = \lambda x : \tau.u$	*then* $s\gamma = \lambda z : \tau.(u\{x \mapsto z\})\gamma$ *with* z *fresh.*
If $s = f(u_1, \ldots, u_n)$	*then* $s\gamma = f(u_1\gamma, \ldots, u_n\gamma)$
If $s = X(u_1, \ldots, u_n)$ *and* $X \notin \mathcal{V}ar(\gamma)$	*then* $s\gamma = X(u_1\gamma, \ldots, u_n\gamma)$
If $s = X(u_1, \ldots, u_n)$ *and* $(X : \sigma) \mapsto (\Gamma, \lambda y_1 : \tau_1 \ldots y_n : \tau_n.u) \in \gamma$	*then* $s\gamma = @(\lambda y_1 : \tau_1 \ldots y_n : \tau_n.u, u_1\gamma, \ldots, u_n\gamma)$

In the last case, we could also perform the introduced beta-reductions therefore hiding the application operator. Writing $s\gamma$ assumes $\mathcal{D}om(\gamma)$ compatible with $\Gamma \vdash_{\mathcal{F}} s : \sigma$ and it follows that $(\Gamma \setminus \mathcal{D}om(\gamma)) \cup \mathcal{R}an(\gamma) \vdash_{\mathcal{F}} s\gamma : \sigma$. Given γ, the *composition* or *instance* $(\{(x_i : \sigma_i) \mapsto (\Gamma_i, t_i)\}_i)\gamma$ is the substitution $\{(x_i : \sigma_i) \mapsto ((\Gamma_i \setminus \mathcal{D}om(\gamma)) \cup \mathcal{R}an(\gamma), t_i\gamma)\}_i$.

3.6 Higher-order rewriting relations

We now introduce the different variants of higher-order rewriting, categorized by their use of pattern matching. Allowing for variables with arities should be considered as a cosmetic variation of the traditional frameworks, since this is the case for the pattern matching and unification algorithms, as well as for the definition of patterns.

Plain higher-order rewriting.

Definition 5. *A* plain higher-order rewrite system *is a set of higher-order rewrite rules* $\{\Gamma_i \vdash l_i \to r_i : \sigma_i\}_i$ *such that*
(i) $\Gamma_i \vdash_{\mathcal{F}} l_i : \sigma_i$ *and* $\Gamma_i \vdash_{\mathcal{F}} r_i : \sigma_i$*,*
(ii) $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$*.*

Plain higher-order rewriting is based on plain pattern matching:

$$\Sigma \vdash u \underset{\Gamma \vdash l \to r}{\xrightarrow{p}} v \text{ if } \Sigma \vdash_{\mathcal{F}} u : \tau \text{ for some type } \tau, u|_p = l\sigma \text{ and } v = u[r\sigma]_p.$$

Note the need for an environment Σ in which u is typable. Then, it is easy to see that $\Sigma \vdash_{\mathcal{F}} v : \tau$, a property called type preservation, which allows us to actually drop the environment Σ so as to make our notations more readable. We also often take the liberty to drop the type of a rule.

A *higher-order equation* is a pair of higher-order rewrite rules $\{\Gamma \vdash l \to r : \sigma, \Gamma \vdash r \to l : \sigma\}$. We abbreviate such a pair by $\Gamma \vdash l = r : \sigma$, and write $\Sigma \vdash u \longleftrightarrow_R^* v$ or $\Sigma \vdash u =_R v$ if R is a set of equations.

Conversion rules. Conversion rules in the typed lambda calculus are a particular example of higher-order equations:

$$\begin{array}{rl} \{V : \tau\} \vdash & \lambda x : \sigma.V =_\alpha \lambda y : \sigma.V\{x \mapsto y\} \\ \{U : \sigma, V : \tau\} \vdash & @(\lambda x : \sigma.V, U) =_\beta V\{x \mapsto U\} \\ \{U : \sigma \to \tau\} \vdash & \lambda x : \sigma.@(U, x) =_\eta U \end{array}$$

Traditionally, these equations are schemas, in which U and V stand for arbitrary terms, with $x \notin \mathcal{V}ar(U)$ for the eta-rule and $y \notin (\mathcal{V}ar(V) \setminus \{x\})$ for the alpha-rule. Here, these conditions are ensured by our definition of substitution, hence these equations are indeed true higher-order equations.

Orienting the last two equalities from left to right yields the beta-reduction (resp. eta-reduction) rule. Orienting the third equation from right to left yields the eta-expansion rule. Since this rule may not terminate, its use is restricted by spelling out in which context it is allowed:

$$\begin{array}{c} \{u : \sigma_1 \to \ldots \to \sigma_n \to \sigma\} \vdash \\ s[u]_p \longrightarrow_\eta^p s[\lambda x_1 : \sigma_1, \ldots, x_n : \sigma_n.@(u, x_1, \ldots, x_n)]_p \end{array}$$

$$\textbf{if} \begin{cases} \sigma \text{ is a data type}, \quad x_1, \ldots, x_n \notin \mathcal{V}ar(u), \\ u \text{ is not an abstraction and } s|_q \text{ is not an application in case } p = q \cdot 1 \end{cases}$$

The last condition means that the first argument of an application cannot be recursively expanded on top. A variant requires in addition that $p \neq \Lambda$.

Typed lambda-calculi are confluent modulo alpha-conversion, and terminating with respect to beta-reductions and either eta-expansion or eta-reduction, therefore defining normal forms up to the equivalence generated by alpha-conversion. Our notations for normal forms use down arrows for reductions: $u{\downarrow_\beta}$ and $u{\downarrow_{\beta\eta}}$; up arrows for expansions: $u{\uparrow^\eta}$; and their combined version for eta-long beta-normal forms: $u{\updownarrow_\beta^\eta}$. We use $u{\downarrow}$ for a normal form of some unspecified kind.

We can now introduce most general substitutions. Given $\Gamma \vdash_{\mathcal{F}} s, t : \sigma$, a *solution* (resp. *higher-order solution*) of the *equation* $s = t$ is a substitution γ such that $s\gamma = t$ (resp. $s\gamma =_{\beta\eta} t$) for a *plain* (resp. *higher-order*) *pattern-matching* problem, and $s\gamma = t\gamma$ (resp. $s\gamma =_{\beta\eta} t\gamma$) for a *plain* (resp. *higher-order*) *unification* problem. Solutions are called *plain/higher-order matches/unifiers*, depending on which case is considered. A unifier γ is *most general* if any unifier θ satisfies $\theta = \gamma\varphi$ (resp. $\theta =_{\beta\eta} \gamma\varphi$) for some substitution φ.

Normal higher-order rewriting. Normal higher-order rewriting is based upon higher-order pattern matching, rules must be normalized, and their lefthand sides must be patterns. Our definition of patterns specializes to Nipkow's [24] when variables have arity zero:

Definition 6. *A higher-order term u is a pattern if for every variable occurrence of some variable $X : \sigma_1 \to \ldots \sigma_m \Rightarrow \tau_1 \to \ldots \tau_n \in \mathcal{V}ar(u)$ with $n > 1$ and τ_n a data type, there exists a position $p \in \mathcal{P}os(u)$ such that*

(i) $u|_p = @(X(x_1, \ldots, x_m), x_{m+1}, \ldots, x_{m+n})$,

(ii) $\forall i, j \in [1..m+n],\ x_i \neq x_j$ or $i{=}j$,

(iii) $\forall i \in [1..m+n]$, there exists a position $q < p$ in $\mathcal{P}os(u)$ and a term v such that $u|_q = \lambda x_i.v$.

Assuming $X : \alpha \Rightarrow \beta \to \gamma$, $\lambda xy.@(X(x), y)$ is a pattern while $\lambda x.X(x)$, $\lambda x.@(X(x), x)$ and $\lambda xy.Y(@(X(x), y)$ are not.

Computing the normal form of a pattern instance is done by normalizing first the pattern u then the substitution γ, before to reduce every subexpression $@(X\gamma(x_1, \ldots, x_m), x_{m+1}, \ldots, x_{m+n})$. This simple schema in which the third step is a development shows that Nipkow's and Klop's notions of rewriting coincide when lefthand sides of rules are patterns. We believe that this is the very reason why higher-order pattern matching and higher-order unification are decidable for patterns, suggesting a more general notion of pattern. A related observation is made in [13].

Definition 7. *A normal higher-order rewrite system is a set of rules $\{\Gamma_i \vdash l_i \to r_i : \sigma_i\}_i$ such that conditions (i) and (ii) of Definition 5 are satisfied, (iii) l_i and r_i are in normal form and l_i is a pattern.*

Normal higher-order rewriting operates on terms in normal form:

$$\Sigma \vdash u \underset{\Gamma \vdash l \to r}{\xrightarrow{p}} v \text{ if } u = u{\downarrow}, u|_p \underset{\beta\eta}{\overset{*}{\longleftrightarrow}} l\sigma \text{ and } v = u[r\sigma]_p{\downarrow}$$

There are several variants of normal higher-order rewriting, which use different kinds of normal forms.

In [26] and [24], eta-long beta-normal forms are used. The higher-order rewrite rules must satisfy the following additional condition:

(iv) type constructors are constants, there are no type variables, function symbols and variables have arity 0, and rules are of basic type.

In [21], the framework is generalized, and condition (iv) becomes:

(v) type constructors may have a non-zero arity, type variables are allowed, function symbols have aritypes, variables have arity 0, terms

are assumed to be in eta-long beta-normal form except at the top (eta-expansion is not applied at the top of rules, but beta-normalization is), rules can be of any (possibly polymorphic) type.

In [22], beta-eta-normal forms are used, and condition (iv) becomes:

(vi) type constructors have arities, type variables are allowed, constants and variables have aritypes, lefthand sides of rules are of the form $f(l_1, \ldots, l_n)$ for some function symbol f, and rules are of any type.

The use of beta-eta-normal forms allows us to restrict the lefthand sides of rules by assuming they are headed by a function symbol. Were eta-long beta-normal forms used instead, this assumption would imply Nipkow's restriction that rules are of basic type. We will see some advantage of beta-eta-normal forms for proving confluence and termination.

Mixed higher-order rewriting. There is no reason to restrict ourselves to a single kind of higher-order rules. It is indeed possible to use both, by pairing each rule with the pattern-matching algorithm it uses, as done with the keywords Prr, Hor used in the examples of Section 2. Of course, we need to have all lefthand sides of both kinds of rules to be patterns, and need using the same kind of normal form when rewriting terms. The choice of beta-eta-normal forms is dictated by its advantages.

One can wonder whether there is a real need for two different keywords for higher-order rules. As we have seen, higher-order pattern matching is needed if and only if beta-redexes can be created by instantiation at non-variable positions of the lefthand sides (eta-redexes cannot). This requires a subterm $F(u_1, \ldots, u_m)$ in the lefthand side of the rule, where F is a free variable of aritype $\sigma_1 \to \ldots \sigma_m \Rightarrow \sigma$ with $m > 0$, an easily decidable property.

Mixed higher-order rewriting has not yet been studied in the literature although it is explicitely alluded to in [22]. However, all known results can be easily lifted to the mixed case. In the sections to come, we will present the results known for both plain and higher-order rewriting, and formulate their generalization as a conjecture when appropriate.

4 Confluence

In this section, we restrict our attention to terminating relations. Confluence will therefore be checked via critical pairs, whose kind is imposed by the mecanism for searching redexes.

Definition 8. *Given two rules $l \to r$ and $g \to d$, a non-variable position $p \in \mathcal{P}os(l)$ and a most general unifier (resp. most general higher-order*

unifier) σ *of the equation* $l|_p = g$*, the pair* $(r\sigma, l[d\sigma]_p)$ *is called a* critical pair *(resp., a* higher-order critical pair*) of* $g \to d$ *on* $l \to r$ *at position* p*.*

A critical pair (s,t) *is* joinable *if there exists* v *such that* $s \longrightarrow_R^* v$ *and* $t \longrightarrow_R^* v$*. It is* higher-order joinable *(*joinable *when clear from the context) if there exist* v, w *such that* $s \longrightarrow_R^* v, t \longrightarrow_R^* w$ *and* $v \longleftrightarrow_{\beta\eta}^* w$*.*

Plain higher-order rewriting. Using plain pattern matching leads to plain critical pairs computed with plain unification. The following result follows easily from Newman's Lemma [15]:

Theorem 1. *[7]. Given a set* R *of higher-order rules such that*
(a) $R \cup \{beta, eta\}$ *is terminating;*
(a) all critical pairs in $R \cup \{beta,$ *eta*$\}$ *are joinable;*
then plain higher-order rewriting with $R \cup \{beta, eta\}$ *is confluent.*

It can easily be checked that Examples 1, 2 and 3 are confluent because they do not admit any critical pair.

Higher-order rewriting. Replacing joinability by higher-order joinability does not allow us to get a similar result for higher-order rewriting [24]. It is indeed well-known, in the first-order case, that the natural generalization of *confluence* of a set of rules R in presence of a set of equations E:

$$\begin{array}{c} \forall s,t,u \text{ in normal form such that } u \longrightarrow_R^* s \text{ and } u \longrightarrow_R^* t \\ \exists v,w \text{ such that } s \longrightarrow_R^* v, t \longrightarrow_R^* w \text{ and } v \longleftrightarrow_E^* w \end{array}$$

is not enough for ensuring the *Church-Rosser property*:

$$\begin{array}{c} \forall s,t \text{ such that } s \longleftrightarrow_{R\cup E}^* t \\ \exists v,w \text{ such that } s \longrightarrow_R^* v, t \longrightarrow_R^* w \text{ and } v \longleftrightarrow_E^* w \end{array}$$

when searching for a redex uses E-matching. An additionnal *coherence* property is needed:

$$\begin{array}{c} \forall s,t,u \text{ such that } u \longrightarrow_R^* s \text{ and } u \longleftrightarrow_E^* t \\ \exists v \text{ and } w \text{ such that } s \longrightarrow_R^* v, t \longrightarrow_R^* w \text{ and } v \longleftrightarrow_E^* w. \end{array}$$

Coherence can be ensured for an arbitrary equational theory E by using so-called Stickel's extension rules [29, 16]. In the case of higher-order rewriting, E is made of alpha, beta and eta. Then, rules headed by an abstraction operator on the left, that is, of the form

$$\lambda x.l \to r \quad \text{need as beta-extension the rule} \quad l \to @(r,x)\downarrow$$

obtained by putting both sides of $\lambda x.l \to r$ inside $@([], x)$, the minimal context generating a beta redex on top of its lefthand side. Normalizing the result yields the extension. Note that a single extension is generated.

For example, the rule $\lambda x.a \to \lambda x.b$, where a, b are constants has $a \to b$ as extension. Indeed, $a \longleftarrow^{\Lambda}_{\beta} @(\lambda x.a, x) \longrightarrow^{1}_{\lambda x.a \to \lambda x.b} @(\lambda x.b, x) \longrightarrow^{\Lambda}_{\beta} b$. However, $a \neq_{\beta\eta} \lambda x.a$ since a and $\lambda x.a$ have different types. Therefore, a and b are different terms in normal-form, although they are equal in the theory generated by eta, beta, and the equation $\lambda x.a = \lambda x.b$. Adding the extension $a \to b$ solves the problem.

This explains Nipkow's restriction that rules must be of basic type: in this case, no lefthand side of rule can be an abstraction. Generalizing Nipkow's framework when this assumption is not met is not hard: it suffices to close the set of rules with finitely many beta-extensions for those rules whose lefthand side is an abstraction. This covers rules of arbitrary polymorphic functional type. Notice also that no beta-extension is needed for rules whose lefthand side is headed by a function symbol. Finally, because of the pattern condition, it is easy to see that no extension is needed for the eta-rule. We therefore have the following result:

Theorem 2. *[22] Given a set R of higher-order rules satisfying assumptions (i,ii,iii) and (v) such that*

(a) $R \cup \{beta, eta^{-1}\}$ is terminating;

(b) R is closed under the computation of beta-extensions;

(c) irreducible higher-order critical pairs of R are joinable;

then, higher-order rewriting with R is Church-Rosser.

Nipkow's result stating confluence of higher-order rewriting when assumption (iv) is met appears then as a corollary. The fact that it is a corollary is not straightforward, since the termination assumption is different: Nipkow assumes termination of higher-order rewriting with R. The fact that this coincides with assumption (i) is however true under assumptions (i,ii,iii,iv) [22]. We can easily see that Examples 4 and 5 (first version) do not admit higher-order critical pairs, hence are Church-Rosser. Adding the other rules for differentiation would clearly not change this situation.

Adding beta-extensions is not such a burden, but we can even dispense with by using the variant of eta-long beta-normal forms in which eta-expansion does not apply at the top of terms. Then, we can assume that the lefthand side of a higher-order rule is headed by an application or by a function symbol of non-zero arity if they are allowed, and no beta-extension is needed anymore.

We now turn to the second kind of normal form:

Theorem 3. *[22] Given a set R of higher-order rules satisfying assumptions (i,ii,iii) and (vi) such that*

(a) $R \cup \{beta, eta\}$ is terminating;

(b) irreducible higher-order critical pairs of R are joinable;

then, higher-order rewriting with R is Church-Rosser.

Example 5, as modified at the end of its paragraph, has no higher-order critical pairs, hence is Church-Rosser.

Altogether, the framework based on $\beta\eta$-normal forms appears a little bit more appealing.

Mixed higher-order rewriting. We end up with the general case of a set of higher-order rules R split into disjoint subsets R_1 using plain pattern matching, and R_2 using higher-order pattern matching for terms in beta-eta-normal form. We assume that R satisfies assumptions (i,ii), that lefthand sides of rules in R_1 are headed by a function symbol, and that R_2 satisfies assumption (iii,vi). When these assumptions are met, we say that R is a mixed set of higher-order rules. Rewriting then searches for R_1-redexes with plain pattern-matching, and uses beta-eta-normalization before to search for R_2-redexes with higher-order pattern matching.

Our conjecture follows a similar analysis made for the first-order case, with left-linear rules using plain pattern matching, and non-left-linear ones using pattern matching modulo some first-order theory E [16]:

Conjecture. *Given a mixed set $R = R_1 \uplus R_2$ of higher-order rules such that*

(i) $R \cup \{\beta\eta\}$ is terminating;

(ii) irreducible plain critical pairs of R_1 are joinable;

(iii) irreducible higher-order critical pairs of R_2 are joinable;

(iv) irreducible higher-order critical pairs of R_2 with R_1 are joinable;

then, mixed higher-order rewriting with R is Church-Rosser.

5 Termination of plain higher-order rewriting

Given a rewrite system R, a term t is *strongly normalizing* if there is no infinite sequence of rewrites with R issuing from t. R is *strongly normalizing* or *terminating* if every term is strongly normalizing.

Termination of typed lambda calculi is notoriously difficult. Termination of the various incarnations of higher-order rewriting turns out to be even more difficult. There has been little success in coming up with general methods. The most general results have been obtained by Blanqui [5, 6], as a generalization of a long line of work initiated by Jouannaud and

Okada [18, 17, 1, 4, 7]. We will not describe these results here, but base our presentation on the more recent work of Jouannaud and Rubio [19, 20] which is strictly more general in the absence of dependent types. The case of dependent types is investigated in [33], but requires more work.

Higher-order reduction orderings. The purpose of this section is to define the kind of ordering needed for plain higher-order rewriting. To a quasi-ordering $\succeq$, we associate its strict part $\succ$ and equivalence $\simeq$. Typing environments are usually omitted to keep notations simple.

Definition 9. *[20] A* higher-order reduction ordering *is a quasi-ordering $\succeq$ of the set of higher-order terms, which (i) well-founded, (ii) monotonic (i.e., $s \succ t$ implies $u[s] \succ u[t]$ for all contexts $u[]$), (iii) stable (i.e., $s \succ t$ implies $s\gamma \succ t\gamma$ for all compatible substitutions γ) and (iv) includes alpha-conversion (i.e. $=_\alpha \subseteq \simeq$) and beta-eta-reductions (i.e., $\longrightarrow_{\beta\eta} \subset \succ$). It is* polymorphic *if $s \succ t$ implies $s\xi \succ t\xi$ for any type instantiation ξ.*

Note that the above definition includes the eta-rule, and not only the beta-rule as originally. This extension does not raise difficulties.

Theorem 4. *[20] Assume that $\succeq$ is a higher-order reduction ordering, and let $R = \{\Gamma_i \vdash l_i \to r_i\}_{i\in I}$ be a plain higher-order rewrite system such that $l_i \succ r_i$ for every $i \in I$. Assume in addition that $\succeq$ is polymorphic if so is R. Then the relation $\longrightarrow_R \cup \longrightarrow_{\beta\eta}$ is terminating.*

The proof of the previous result is not difficult: it is based on lifting the property $l_i \succ r_i$ to a rewrite step $u \longrightarrow^p_{l_i\to r_i} v$ by using the properties of the ordering. A simple induction allows then to conclude.

Higher-order recursive path ordering. We give here a generalization of Derhowitz's recursive path ordering to higher-order terms. Although the obtained relation is transitive in many particular cases, it is not in general, hence is not an ordering. We will nevertheless call it the *higher order recursive path ordering*, keeping Dershowitz's name for the generalization. We feel free to do so because the obtained relation is well-founded, hence its transitive closure is a well-founded ordering with the same equivalence, taken here equal to alpha-conversion for simplicity. We are given:

1. a partition $Mul \uplus Lex$ of $\mathcal{F} \cup \mathcal{S}$;
2. a quasi ordering $\geq_{\mathcal{FS}}$ on $\mathcal{F} \cup \mathcal{S}$, the *precedence*, such that
   (a) $>_\mathcal{F}$ is well-founded ($\mathcal{F} \cup \mathcal{S}$ is not assumed to be finite);
   (b) if $f =_{\mathcal{FS}} g$, then $f \in Lex$ iff $g \in Lex$;

(c) $>_{\mathcal{FS}}$ is extended to $\mathcal{F} \cup \mathcal{X} \cup \mathcal{S}$ by adding all pairs $x \geq_{\mathcal{FS}} x$ for $x \in \mathcal{X}$ (free variables are only comparable to themselves);

3. a set of terms $\mathcal{CC}(s)$ called the *computability closure* of s: in the coming definition, $\mathcal{CC}(f(\overline{s})) = \overline{s}$, but will become a richer set later on.

The definition compares types and terms in the same recursive manner, using the additional proposition (assuming that s and $f(\overline{t})$ are terms)

$$A = \forall v \in \overline{t} \quad s \underset{horpo}{\succ} v \text{ or } u \underset{horpo}{\succeq} v \text{ for some } u \in \mathcal{CC}(s)$$

Definition 10.

$$s : \sigma \underset{horpo}{\succ} t : \tau \quad \textit{iff} \quad (\sigma = \tau = * \textit{ or } \sigma \underset{horpo}{\succeq} \tau) \textit{ and}$$

1. $s = f(\overline{s})$ *with* $f \in \mathcal{F} \cup \mathcal{S}$, *and* $u \underset{horpo}{\succeq} t$ *for some* $u \in \mathcal{CC}(s)$
2. $s = f(\overline{s})$ *and* $t = g(\overline{t})$ *with* $f >_{\mathcal{FS}} g$, *and* A *(see definition below)*
3. $s = f(\overline{s})$ *and* $t = g(\overline{t})$ *with* $f =_{\mathcal{FS}} g \in Mul$ *and* $\overline{s}(\underset{horpo}{\succ})_{mul}\overline{t}$
4. $s = f(\overline{s})$ *and* $t = g(\overline{t})$ *with* $f =_{\mathcal{FS}} g \in Lex$ *and* $\overline{s}(\underset{horpo}{\succ})_{lex}\overline{t}$, *and* A
5. $s = @(s_1, s_2)$ *and* $u \underset{horpo}{\succeq} t$ *for some* $u \in \{s_1, s_2\}$
6. $s = \lambda x : \sigma.u$, $x \notin \mathcal{V}ar(t)$ *and* $u \underset{horpo}{\succeq} t$
7. $s = f(\overline{s})$, $@(\overline{t})$ *a left-flattening of* t, *and* A
8. $s = f(\overline{s})$ *with* $f \in \mathcal{F}$, $t = \lambda x : \alpha.v$ *with* $x \notin \mathcal{V}ar(v)$ *and* $s \underset{horpo}{\succeq} v$
9. $s = @(s_1, s_2)$, $@(\overline{t})$ *a left-flattening of* t *and* $\{s_1, s_2\}(\underset{horpo}{\succ})_{mul}\overline{t}$
10. $s = \lambda x : \alpha.u$, $t = \lambda x : \beta.v$, $\alpha \underset{horpo}{\simeq} \beta$ *and* $u \underset{horpo}{\succ} v$
11. $s = @(\lambda x.u, v)$ *and* $u\{x \mapsto v\} \underset{horpo}{\succeq} t$

11bis. $s = \lambda x.@(u, x)$ *with* $x \notin \mathcal{V}ar(u)$ *and* $u \underset{horpo}{\succeq} t$

12. $s = \alpha \to \beta$, *and* $\beta \underset{horpo}{\succeq} t$
13. $s = \alpha \to \beta$, $t = \alpha' \to \beta'$, $\alpha \underset{horpo}{\simeq} \alpha'$ *and* $\beta \underset{horpo}{\succ} \beta'$
14. $s = X(\overline{s})$ *and* $t = X(\overline{t})$ *and* $\overline{s}(\succ_{horpo})_{mul}\overline{t}$

We assume known how to extend a relation on a set to n-tuples of elements of the set (*monotonic* extension -comparing terms one by one with at least one strict comparison-, or *lexicographic* extension -comparing terms one by one until a strict comparison is found) or to multisets of elements of the set (*multiset* extension).

The computability closure was called computational closure in [19].

Case 14 does not exist as such in [20], but follows easily from Case 9. Neither does Case 11bis, used for proving the eta-rule. An immediate subterm u of s is *type-decreasing* if the type of u is smaller or equal to the type of s. Condition A differs from its first-order version $\forall v \in \overline{t}\ s \succ_{horpo} v$, but reduces to it when all immediate subterms of s are type-decreasing, because $\succ_{horpo}$ enjoys the subterm property for these subterms. Indeed, the restriction of the ordering to subterm-closed sets of terms whose all immediate subterms are type-drecreasing, as are types, is transitive.

A more abstract description of the higher-order recursive path ordering is given in [20], in which the relation on terms and the ordering on types are separated, the latter being specified by properties to be satisfied. The version given here, also taken from [20], shows that the same mecanism can apply to terms and types, a good start for a generalization to dependent type structures.

Theorem 5. $(\succ_{horpo})*$ *is a polymorphic higher-order reduction ordering.*

The proof is based on Tait and Girard's computability predicate technique (Girard's candidates are not needed for weak polymorphism) [20].

We go on checking the examples of Section 2, making appropriate choices when using property A.

The higher-order recursive path ordering at work.

Example 1. Let us first recall the rules for rec:

$$\mathsf{rec}(\mathsf{0}, \mathsf{U}, \mathsf{Y}) \to U$$
$$\mathsf{rec}(\mathsf{s}(\mathsf{X}), \mathsf{U}, \mathsf{Y}) \to @(Y, X, \mathsf{rec}(\mathsf{X}, \mathsf{U}, \mathsf{Y}))$$

We assume a multiset status for rec.

Since both sides of the first equation have the same (polymorphic) type α, the comparison $\mathsf{rec}(\mathsf{0}, \mathsf{U}, \mathsf{Y}) \succ_{horpo} \mathsf{U}$ proceeds by Case 1 and succeeds easily since Y belongs to the computability closure of $\mathsf{rec}(\mathsf{0}, \mathsf{U}, \mathsf{Y})$ as one of its subterms.

As it can be expected, both sides of the second equation have again the same type. The comparison $\mathsf{rec}(\mathsf{s}(\mathsf{X}), \mathsf{U}, \mathsf{Y}) \succ_{horpo} @(\mathsf{Y}, \mathsf{X}, \mathsf{rec}(\mathsf{X}, \mathsf{U}, \mathsf{Y}))$ proceeds this time by Case 7, generating three subgoals (we use here property A, chosing a subterm when possible). The first subgoal $\mathsf{Y} \succeq_{horpo} \mathsf{Y}$ is solved readily since Y is a subterm of the lefthand side. The second subgoal $\mathsf{s}(\mathsf{X}) \succeq_{horpo} \mathsf{X}$ is solved by Case 1. The third subgoal, namely $\mathsf{rec}(\mathsf{s}(\mathsf{X}), \mathsf{U}, \mathsf{Y}) \succ_{horpo} \mathsf{rec}(\mathsf{X}, \mathsf{U}, \mathsf{Y})$ is solved by Case 3, which generates the three easy comparisons $\mathsf{s}(\mathsf{X}) \succ_{horpo} \mathsf{X}$, $\mathsf{U} \succeq_{horpo} \mathsf{U}$ and $\mathsf{Y} \succeq_{horpo} \mathsf{Y}$.

Example 2. Let us recall the rules:

$$\begin{aligned}\mathsf{map}(\mathsf{nil},\mathsf{F}) &\to \mathsf{nil}\\ \mathsf{map}(\mathsf{cons}(\mathsf{H},\mathsf{T}),\mathsf{F}) &\to \mathsf{cons}(@(\mathsf{F},\mathsf{H}),\mathsf{map}(\mathsf{T},\mathsf{F}))\end{aligned}$$

We assume the precedence $\mathsf{map} >_{\mathcal{FS}} \mathsf{cons}$ and membership of map to Mul. The first rule being easy, let us consider the second. Our goal is
(i) $\mathsf{map}(\mathsf{cons}(\mathsf{H},\mathsf{T}),\mathsf{F}) \succ_{horpo} \mathsf{cons}(@(\mathsf{H},\mathsf{F}),\mathsf{map}(\mathsf{T},\mathsf{F}))$
which first checks types and then reduces to two new goals by Case 2:
(ii) $\mathsf{map}(\mathsf{cons}(\mathsf{H},\mathsf{T}),\mathsf{F}) \succ_{horpo} @(\mathsf{F},\mathsf{H})$
(iii) $\mathsf{map}(\mathsf{cons}(\mathsf{H},\mathsf{T}),\mathsf{F}) \succ_{horpo} \mathsf{map}(\mathsf{T},\mathsf{F})$
Goal (ii) reduces by Case 7 to three new goals
(iv) $\mathsf{list}(\beta) \succ_{horpo} \beta$
(v) $\mathsf{F} \succeq_{horpo} \mathsf{F}$, which disappears
(vi) $\mathsf{cons}(\mathsf{H},\mathsf{T}) \succeq_{horpo} \mathsf{H}$
Goal (iv) is taken care of by Case 1 while Goal (vi) reduces by Case 1 to
(vii) $\mathsf{list}(\alpha) \succ_{horpo} \alpha$
(viii) $\mathsf{H} \succ_{horpo} \mathsf{H}$, which disappears.
Goal (vii) is taken care of by Case 2. We are left with goal (iii) which reduces by Case 3 to
(iv) $\{\mathsf{cons}(\mathsf{H},\mathsf{T}),\mathsf{F}\}(\succ_{horpo})_{mul}\{\mathsf{T},\mathsf{F}\}$
which yields one (last) goal by definition of $(\succ_{horpo})_{mul}$:
(x) $\mathsf{cons}(\mathsf{H},\mathsf{T}) \succ_{horpo} \mathsf{T}$, which is taken care of by Case 1.

Example 3. The example of parametric insertion sort raises a difficulty. Let us recall the rules:

$$\begin{aligned}\mathsf{max}(0,\mathsf{X}) &\to \mathsf{X} \qquad \mathsf{max}(\mathsf{X},0) \to \mathsf{X}\\ \mathsf{max}(\mathsf{s}(\mathsf{X}),\mathsf{s}(\mathsf{Y})) &\to \mathsf{s}(\mathsf{max}(\mathsf{X},\mathsf{Y}))\\ \mathsf{min}(0,\mathsf{X}) &\to 0 \qquad \mathsf{min}(\mathsf{X},0) \to 0\\ \mathsf{min}(\mathsf{s}(\mathsf{X}),\mathsf{s}(\mathsf{Y})) &\to \mathsf{s}(\mathsf{min}(\mathsf{X},\mathsf{Y}))\\ \mathsf{insert}(\mathsf{N},\mathsf{nil},\mathsf{X},\mathsf{Y}) &\to \mathsf{cons}(\mathsf{N},\mathsf{nil})\\ \mathsf{insert}(\mathsf{N},\mathsf{cons}(\mathsf{M},\mathsf{T}),\mathsf{X},\mathsf{Y}) &\to \mathsf{cons}(@(\mathsf{X},\mathsf{N},\mathsf{M}),\mathsf{insert}(@(\mathsf{Y},\mathsf{N},\mathsf{M}),\mathsf{T},\mathsf{X},\mathsf{Y}))\\ \mathsf{sort}(\mathsf{nil},\mathsf{X},\mathsf{Y}) &\to \mathsf{nil}\\ \mathsf{sort}(\mathsf{cons}(\mathsf{N},\mathsf{T}),\mathsf{X},\mathsf{Y}) &\to \mathsf{insert}(\mathsf{N},\mathsf{sort}(\mathsf{T},\mathsf{X},\mathsf{Y}),\mathsf{X},\mathsf{Y})\\ \mathsf{ascending\_sort}(\mathsf{L}) &\to \mathsf{sort}(\mathsf{L},\lambda\mathsf{xy}.\mathsf{min}(\mathsf{x},\mathsf{y}),\lambda\mathsf{xy}.\mathsf{max}(\mathsf{x},\mathsf{y}))\\ \mathsf{descending\_sort}(\mathsf{L}) &\to \mathsf{sort}(\mathsf{L},\lambda\mathsf{xy}.\mathsf{max}(\mathsf{x},\mathsf{y}),\lambda\mathsf{xy}.\mathsf{min}(\mathsf{x},\mathsf{y}))\end{aligned}$$

The reader can check that all comparisons succeed with appropriate precedence and statuses, but the last two:

$$\begin{aligned}\mathsf{ascending\_sort}(\mathsf{L}) &\succ_{horpo} \mathsf{sort}(\mathsf{L},\lambda\mathsf{xy}.\mathsf{min}(\mathsf{x},\mathsf{y}),\lambda\mathsf{xy}.\mathsf{max}(\mathsf{x},\mathsf{y}))\\ \mathsf{descending\_sort}(\mathsf{L}) &\succ_{horpo} \mathsf{sort}(\mathsf{L},\lambda\mathsf{xy}.\mathsf{max}(\mathsf{x},\mathsf{y}),\lambda\mathsf{xy}.\mathsf{min}(\mathsf{x},\mathsf{y}))\end{aligned}$$

This is because the subterm $\lambda\mathsf{xy.min(x,y)}$ occuring in the righthand side has type $\alpha \to \alpha \to \alpha$, which is not comparable to any lefthand side type.

Computability closure Our definition of $\succ_{horpo}$ is parameterized by the definition of the computability closure. In order for Theorem 5 to hold, it suffices to make sure that the closure is made of terms which are all computable in the sense of Tait and Girard's computability predicate (termed *reducibility candidate* by Girard). Computability is guaranteed for the immediate subterms of a term by an induction argument in the strong normalization proof, and we can then enlarge the set of computable terms by using computability preserving operations.

Definition 11. *Given a term $t = f(\overline{t})$ of type σ, we define its computable closure $\mathcal{CC}(t)$ as $\mathcal{CC}(t, \emptyset)$, where $\mathcal{CC}(t, \mathcal{V})$, with $\mathcal{V} \cap \mathcal{V}ar(t) = \emptyset$, is the smallest set of well-typed terms containing all variables in $\mathcal{V}$, all terms in $\overline{t}$, and closed under the following operations:*

1. *subterm of minimal data-type: let $s \in \mathcal{CC}(t, \mathcal{V})$ with $f \in \mathcal{F}$, and $u : \sigma$ be a subterm of s such that σ is a data-type minimal in the type ordering and $\mathcal{V}ar(u) \subseteq \mathcal{V}ar(t)$; then $u \in \mathcal{CC}(t, \mathcal{V})$;*
2. *precedence: let g such that $f >_{\mathcal{FS}} g$, and $\overline{s} \in \mathcal{CC}(t, \mathcal{V})$; then $g(\overline{s}) \in \mathcal{CC}(t, \mathcal{V})$;*
3. *recursive call: let $\overline{s}$ be a sequence of terms in $\mathcal{CC}(t, \mathcal{V})$ such that the term $f(\overline{s})$ is well typed and $\overline{t}(\succ_{horpo} \cup \rhd\succ)_{stat_f} \overline{s}$; then $f(\overline{s}) \in \mathcal{CC}(t, \mathcal{V})$;*
4. *application: let $s : \sigma_1 \to \ldots \to \sigma_n \to \sigma \in \mathcal{CC}(t, \mathcal{V})$ and $u_i : \sigma_i \in \mathcal{CC}(t, \mathcal{V})$ for every $i \in [1..n]$; then $@(s, u_1, \ldots, u_n) \in \mathcal{CC}(t, \mathcal{V})$;*
5. *abstraction: let $x \notin \mathcal{V}ar(t) \cup \mathcal{V}$ and $s \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$; then $\lambda x.s \in \mathcal{CC}(t, \mathcal{V})$;*
6. *reduction: let $u \in \mathcal{CC}(t, \mathcal{V})$, and $u \succeq_{horpo} v$; then $v \in \mathcal{CC}(t, \mathcal{V})$;*
7. *weakening: let $x \notin \mathcal{V}ar(u, t) \cup \mathcal{V}$. Then, $u \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$ iff $u \in \mathcal{CC}(t, \mathcal{V})$.*

The new definition of the computability closure uses the relation $\succ_{horpo}$, while the new relation is denoted by $\succ_{chorpo}$. Whether this notational difference introduced in [19] is necessary is doubtful, but allows to define the ordering in a hierachical way rather than as a fixpoint. The proofs would otherwise be surely harder than they already are.

We can now show termination of the two remaining rules of Example 3:
(i) $\mathsf{ascending\_sort(L)} \succ_{chorpo} \mathsf{sort(L, \lambda xy.min(x,y), \lambda xy.max(x,y))}$
(ii) $\mathsf{descending\_sort(L)} \succ_{chorpo} \mathsf{sort(L, \lambda xy.max(x,y), \lambda xy.min(x,y))}$

Since both proofs are almost identical, we only consider goal (i). We assume the precedence $\mathsf{ascending\_sort} >_{\mathcal{FS}} \mathsf{sort, min, max}$. First, note that

using rule 2 does not work, since we would generate the goal
$\mathsf{ascending\_sort}(\mathsf{L}) \succ_{chorpo} \lambda xy.max(x,y)$
which fails for typing reason. Instead, we immediately proceed with Case 1 of the ordering definition, showing that the whole righthand side is in the computability closure of the lefthand one:
(iii) $\mathsf{sort}(\mathsf{L}, \lambda\mathsf{xy.max(x,y)}, \lambda\mathsf{xy.min(x,y)}) \in \mathcal{CC}(\mathsf{ascending\_sort}(\mathsf{L}), \emptyset)$
By precedence Case 2 of the computability closure, we get three goals:
(iv) $\mathsf{L} \in \mathcal{CC}(\mathsf{ascending\_sort}(\mathsf{L}), \emptyset)$
(v) $\lambda\mathsf{xy.max(x,y)} \in \mathcal{CC}(\mathsf{ascending\_sort}(\mathsf{L}), \emptyset)$
(vi) $\lambda\mathsf{xy.min(x,y)}) \in \mathcal{CC}(\mathsf{ascending\_sort}(\mathsf{L}), \emptyset)$
Goal (iv) is easily done by the basic case of the inductive definition of the computability closure. Goal (v) is similar to goal (vi), which we do now. By weakening Case 7 applied twice, we get the new goal:
(vii) $\mathsf{min(x,y)}) \in \mathcal{CC}(\mathsf{ascending\_sort}(\mathsf{L}), \{\mathsf{x},\mathsf{y}\})$
Applying now precedence Case 2 of the closure, we get the new goals
(viii) $x \in \mathcal{CC}(\mathsf{ascending\_sort}(\mathsf{L}), \{x,y\})$
(ix) $y \in \mathcal{CC}(\mathsf{ascending\_sort}(\mathsf{L}), \{x,y\})$
which are both solved by basic case of the inductive definition.

6 Termination of normal higher-order rewriting

The situation with normal higher-order rewriting is a little bit more delicate because of the definition in which rewritten terms are normalized. Lifting a comparison from a rule $l \succ r$ to an instance $l\sigma\downarrow \succ r\sigma\downarrow$ may not be possible for a given higher-order reduction ordering, but may be for some appropriate subrelation.

Definition 12. *A subrelation $\succ_\beta$ of $\succ$ is said to be* $\downarrow$-stable *if $s \succ_\beta t$ implies $s\gamma\downarrow_\beta \succ t\gamma\downarrow_\beta$ for all normalized terms s,t and normalized substitution γ.*

Theorem 6. *[20] Assume that $\succ$ is a higher-order reduction ordering and that $\succ_\beta$ is a $\downarrow$-stable subrelation of $\succ$. Let $R = \{\Gamma_i \vdash l_i \to r_i\}_{i\in I}$ be a normal higher-order rewrite system such that $l_i \succ_\beta r_i$ for every $i \in I$. Assume in addition that $\succeq$ is polymorphic if so is R. Then normal higher-order rewriting with R is strongly normalizing.*

The proof of this result is quite similar to that of Theorem 4. Let us remark that $\succ_{horpo}$ is not $\downarrow$-stable. The following example shows the kind of problematic term:

Example 8. $@(X, f(a)) \succ_{horpo} @(X, a) \succ_{horpo} a$, while instantiating these comparisons with the substitution $\gamma = \{X \mapsto \lambda y.a\}$ yields $X(f(a))\gamma\downarrow \;=\; X(a)\gamma\downarrow \;=\; a$, contradicting $\downarrow$-stability. □

It turns out that only the beta rule may cause a problem for the higher-order recursive path ordering, hence our notation $\succ_\beta$, by turning a greater than comparison into an greater than or equal to comparison. As a consequence, the coming discussion applies to all kinds of normal higher-order rewriting, using eta either as a reduction or as an expansion, and with or without arities.

Definition 13. *The relation* $(\succ_{horpo})_\beta$ *, called* normal higher-order recursive path ordering *is defined as the relation* $\succ_{horpo}$*, but restricting Cases 5 and 9 as follows:*

5. *... if* s_1 *is not an abstraction nor a variable, and* $u = s_2$ *otherwise.*
9. *... if* s_1 *is not an abstraction nor a variable, and* $\{s_2\}((\succ_{horpo})_\beta)_{mul}\overline{t}$ *otherwise.*

Although the main argument of the proof is that this definition yields a (well-founded) subrelation of the higher-order recursive path ordering, showing that it is beta-stable happens to be a painful technical task, especially when using eta-expansions. Note that we could think adding two new cases, by incorporating the cases removed from $(\succ_{horpo})_\beta$ in the equivalence part of the new relation:

14. $@(X(\overline{s}), \overline{u})(\succeq_{horpo})_\beta\ @(X(\overline{t}), \overline{v})$ *if* $\overline{s}\,\overline{u}((\succeq_{horpo})_\beta)_{mon}\overline{t}\overline{v}$

The study of this variant should be quite easy.

It is unfortunately not enough to modify the ordering as done here in presence of a computational closure which is not reduced to the set of subterms: the definition of the closure itself needs cosmetic modifications to ensure that the ordering is stable. Since they do not impact the example to come, we simply refer to the original article for its precise definition [21]. Let us denote the obtained relation by $(\succ_{chorpo})_\beta$.

Theorem 7. $((\succ_{horpo})_\beta)^*$ *and* $((\succ_{chorpo})_\beta)^*$ *are polymorphic* $\downarrow$*-stable higher-order reduction orderings.*

We now test the ordering $(\succ_{chorpo})_\beta$ against our remaining examples. We shall refer to Definition 10 for the cases of Definition 13 that did not change, and to the modifications for the changed ones.

Example 4. **Differentiation 1.** Let us recall the goal, writing all applications on left, and flattening them all on the right:
(i) $@(@(\mathsf{diff}, \lambda\mathsf{x}.\, @(\mathsf{sin}, (@(\mathsf{F}, \mathsf{x})))), \mathsf{y})$ $(\succ_{chorpo})_\beta$
$@(\mathsf{mul}, @(\mathsf{cos}, @(\mathsf{F}, \mathsf{y})), @(\mathsf{diff}, \lambda\mathsf{x}.@(\mathsf{F}, \mathsf{x}), \mathsf{y}))$

We take $D >_\mathcal{F} \{mul, sin, cos\}$ for precedence and assume that the function symbols are in Mul. By Case 9 applied to the goal (i), we get
(ii) $\{@(\mathsf{diff}, \lambda\mathsf{x}.\, @(\mathsf{sin}, (@(\mathsf{F}, \mathsf{x})))), \mathsf{y}\}((\succ_{chorpo})_\beta)_{mul}$
$\{\mathsf{mul}, @(\mathsf{cos}, @(\mathsf{F}, \mathsf{y})), @(\mathsf{diff}, \lambda\mathsf{x}.@(\mathsf{F}, \mathsf{x}), \mathsf{y})\}$
Because y occurs free in the term $@(\mathsf{cos}, @(\mathsf{F}, \mathsf{y}))$ taken from the multiset on the right and because y is the only term in the left multiset in which y occurs free, there is no possibility to solve the previous goal and the whole comparison fails. There might be a way out by using the closure mechanism for applications, but this variant has not been studied yet.

Example 5. **Differentiation 2.** Let us recall this applications-free goal:
(i) $\mathsf{diff}(\lambda\mathsf{x}.\, \mathsf{sin}(\mathsf{F}(\mathsf{x})))(\succ_{chorpo})_\beta\ \mathsf{mul}(\lambda\mathsf{x}.\, \mathsf{cos}(\mathsf{F}(\mathsf{x})), \mathsf{diff}(\lambda\mathsf{x}.\mathsf{F}(\mathsf{x})))$
We take the precedence $\mathsf{diff} >_{\mathcal{FS}} \mathsf{mul} >_{\mathcal{FS}} \mathsf{sin} =_{\mathcal{FS}} \mathsf{cos}$ and assume that all function symbols are in Mul. By Case 2 applied to goal (i), we get:
(ii) $\lambda\mathsf{x}.\, \mathsf{sin}(\mathsf{F}(\mathsf{x}))(\succeq_{chorpo})_\beta\ \lambda\mathsf{x}.\, \mathsf{cos}(\mathsf{F}(\mathsf{x}))$
(iii) $\mathsf{diff}(\lambda\mathsf{x}.\, \mathsf{sin}(\mathsf{F}(\mathsf{x})))(\succ_{chorpo})_\beta\ \mathsf{diff}(\lambda\mathsf{x}.\mathsf{F}(\mathsf{x}))$
Goal (ii) reduces successively to
(iv) $\mathsf{sin}(\mathsf{F}(\mathsf{x}))(\succeq_{chorpo})_\beta\ \mathsf{cos}(\mathsf{F}(\mathsf{x}))$ by Case 10
(v) $\mathsf{F}(\mathsf{x})(\succeq_{chorpo})_\beta\ \mathsf{F}(\mathsf{x})$ by Case 3, which disappears.
We proceed with goal (iii) which reduces successively to
(vi) $\lambda\mathsf{x}.\, \mathsf{sin}(\mathsf{F}(\mathsf{x}))(\succ_{chorpo})_\beta\ \lambda\mathsf{x}.\mathsf{F}(\mathsf{x})$ by Case 3
(viii) $\mathsf{sin}(\mathsf{F}(\mathsf{x}))(\succ_{chorpo})_\beta\ \mathsf{F}(\mathsf{x})$ by Case 10
(ix) $\mathsf{F}(\mathsf{x}) \in \mathcal{CC}(\mathsf{sin}(\mathsf{F}(\mathsf{x})))$ by Case 1,
which disappears by base case of the closure definition.

With the same precedence, the reader can now try the modified version of Example 5: the computation goes as if using Dershowitz's recursive path ordering. We observe the influence of seemingly equivalent definitions on the computation, suggesting that some work is still necessary to improve the higher-order recursive path ordering in the normalized case.

7 Termination of mixed higher-order rewriting

Conjecture. *Given a mixed set $R = R_1 \uplus R_2$ of higher-order rules, a polymorphic higher-order ordering $\succ$ and a $\downarrow_{\beta\eta}$-stable subrelation $\succ_{\beta\eta}$ of $\succ$ such that $l \succ r$ for any rule $l \to r \in R_1$ and $l \succ_{\beta\eta} r$ for any rule $l \to r \in R_2$, then mixed rewriting with R is terminating.*

The proof follows from the fact $\succ$ decreases along beta-eta-reductions.

8 Conclusion

We have shown here how the various versions of higher-order rewriting relate to each other, and can be enriched so as to yield a new framework in which they coexist. We have also shown how the classical properties of rewriting can be checked with the help of powerful tools, like higher-order critical pairs, beta-extensions, the higher-order recursive path ordering, and the computing closure. We left out sufficient completeness, which has been only recently adressed in the case of plain rewriting [8].

Acknowledgments: This paper would not exist without the pionneering work of Klop on higher-order rewriting and the work I have done myself with my coauthors Femke van Raamsdonk and Albert Rubio, my colleague Mitsuhiro Okada with who I started getting into that subject, and my students Maribel Fernandez, Frederic Blanqui and Daria Walukiewicz who investigated some of these questions within the framework of the calculus of constructions. An anonymous referee and Femke van Raamsdonk deserve special thanks for their help in shaping the initial draft.

References

1. Franco Barbanera, Maribel Fernández, and Herman Geuvers. Modularity of strong normalization and confluence in the algebraic-λ-cube. In *Proc. of the 9th Symp. on Logic in Computer Science*, IEEE Computer Society, 1994.
2. Henk Barendregt. *Handbook of Logic in Computer Science*, chapter Typed lambda calculi. Oxford Univ. Press, 1993. eds. Abramsky et al.
3. Terese. Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science 55, Marc Bezem, Jan Willem Klop and Roel de Vrijer eds., Cambridge University Press, 2003.
4. Frédéric Blanqui, Jean-Pierre Jouannaud, and Mitsuhiro Okada. The Calculus of Algebraic Constructions. In Narendran and Rusinowitch, Proc. RTA, LNCS 1631, 1999.
5. Frédéric Blanqui. Inductive Types Revisited. Available from the web.
6. Frédéric Blanqui. Definitions by rewriting in the Calculus of Constructions, 2003. To appear in Mathematical Structures in Computer Science.
7. Frédéric Blanqui, Jean-Pierre Jouannaud, and Mitsuhiro Okada. Inductive Data Types. *Theoretical Computer Science* 277, 2001.
8. Jacek Chrzaczsz and Daria Walukiewicz-Chrzaczsz. Consistency and Completeness of Rewriting in the Calculus of Constructions. Draft.
9. Nachum Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, March 1982.
10. Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–309. North-Holland, 1990.
11. Gilles Dowek. Higher-Order Unification and Matching. Handbook of Automated Reasonning, A. Voronkov ed., vol 2, pages 1009–1062.

12. Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide version 5.6. INRIA Rocquencourt and ENS Lyon.
13. Gilles Dowek, Thérèse Hardin, Claude Kichner and Franck Pfenning. Unification via explicit substitutions: The case of Higher-Order Patterns. In *JICSLP*:259–273, 1996.
14. Jean-Yves Girard, Yves Lafont, and Patrick Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.
15. Gérard Huet. Confluent reductions: abstract properties and applications to term rewriting systems. In *Journal of the ACM* 27:4(797–821), 1980.
16. Jean-Pierre Jouannaud and Hélène Kirchner. Completion of a Set of Rules Modulo a Set of Equations. In *Siam Journal of Computing* 15:4(1155–1194), 1984.
17. Jean-Pierre Jouannaud and Mitsuhiro Okada. Abstract data type systems. *Theoretical Computer Science*, 173(2):349–391, February 1997.
18. Jean-Pierre Jouannaud and Mitsuhiro Okada. Higher-Order Algebraic Specifications. In *Annual IEEE Symposium on Logic in Computer Science*, Amsterdam, The Netherlands, 1991. IEEE Comp. Soc. Press.
19. Jean-Pierre Jouannaud and Albert Rubio. The higher-order recursive path ordering. In Giuseppe Longo, editor, *Fourteenth Annual IEEE Symposium on Logic in Computer Science*, Trento, Italy, July 1999. IEEE Comp. Soc. Press.
20. Jean-Pierre Jouannaud and Albert Rubio. Higher-order recursive path orderings. Available from the web.
21. Jean-Pierre Jouannaud and Albert Rubio. Higher-order orderings for normal rewriting. Available from the web.
22. Jean-Pierre Jouannaud and Albert Rubio and Femke van Raamsdonk. Higher-order Rewriting with Types and Arities. Available from the web.
23. Jan Willem Klop. Combinatory Reduction Relations. Mathematical Centre Tracts 127. Mathematisch Centrum, Amsterdam, 1980.
24. Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192(1):3–29, February 1998.
25. Dale Miller. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. In *Journal and Logic and Computation* 1(4):497–536, 1991.
26. Tobias Nipkow. Higher-order critical pairs. In *6th IEEE Symp. on Logic in Computer Science*, pages 342–349. IEEE Computer Society Press, 1991.
27. Tobias Nipkow, Laurence C. Paulson and Markus Wenzel. Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS 2283, Springer Verlag, 2002.
28. Tobias Nipkow and Christian Prehofer. Higher-Order Rewriting and Equational Reasonning. In *Automated deduction — A basis for Applications. Volume I: Foundations*, Bibel and Schmitt editors. Applied Logic Series 8:399–430, Kluwer, 1998.
29. Gerald E. Peterson and Mark E. Stickel. Complete sets of reductions for some equational theories. In *JACM* 28(2):233–264, 1981.
30. Franck Pfenning. Logic Programming in the LF Logical Framework. In *Logical Frameworks*, Gérard Huet and Gordon D. Plotkin eds., Cambridge University Press, 1991.
31. Femke van Raamsdonk. Confluence and Normalization for Higher-Order Rewrite Systems. phd thesis, Vrije Universiteit, Amsterdam, The Netherlands, 1996.
32. Femke van Raamsdonk. Higher-order rewriting. In [3].
33. Daria Walukiewicz-Chrzaszcz. Termination of rewriting in the Calculus of Constructions. In *Proceedings of the Workshop on Logical Frameworks and Metalanguages, Santa Barbara, California*, 2000.