Termination and confluence of higher-order rewrite systems

Frédéric Blanqui

LRI, Université de Paris-Sud

Bât. 490, 91405 Orsay, France

tel: (33) 1.69.15.42.35 fax: (33) 1.69.15.65.86

Frederic.Blanqui@lri.fr

http://www.lri.fr/~blanqui/

Abstract: In the last twenty years, several approaches to higher-order rewriting have been proposed, among which Klop's Combinatory Rewrite Systems (CRSs), Nipkow's Higher-order Rewrite Systems (HRSs) and Jouannaud and Okada's higher-order algebraic specification languages, of which only the last one considers typed terms. The later approach has been extended by Jouannaud, Okada and the present author into Inductive Data Type Systems (IDTSs). In this paper, we extend IDTSs with the CRS higher-order pattern-matching mechanism, resulting in simply-typed CRSs. Then, we show how the termination criterion developed for IDTSs with first-order pattern-matching, called the General Schema, can be extended so as to prove the strong normalization of IDTSs with higher-order pattern-matching. Next, we compare the unified approach with HRSs. We first prove that the extended General Schema can also be applied to HRSs. Second, we show how Nipkow's higher-order critical pair analysis technique for proving local confluence can be applied to IDTSs.

Appendices A, B and C (proofs) are available from the web page.

1 Introduction

In 1980, after a work by Aczel [1], Klop introduced the Combinatory Rewrite Systems (CRSs) [15, 16], to generalize both first-order term rewriting and rewrite systems with bound variables like Church's λ-calculus.

In 1991, after Miller's decidability result of the pattern unification problem [20], Nipkow introduced Higher-order Rewrite Systems (HRSs) [23] (called Pattern Rewrite Systems (PRSs) in [18]), to investigate the metatheory of logic programming languages and theorem provers like λProlog [21] or Isabelle [25]. In particular, he extended to the higher-order case the decidability result of Knuth and Bendix about local confluence of first-order term rewrite systems.

At the same time, after the works of Breazu-Tannen [6], Breazu-Tannen and Gallier [7] and Okada [24] on the combination of Church's simply-typed λ-calculus with first-order term rewriting, Jouannaud and Okada introduced higher-order algebraic specification languages [11, 12] to provide a computational model for typed functional languages extended with first-order and higher-order rewrite definitions. Later, together with the present author, they extended these languages with (strictly positive) inductive types, leading to Inductive Data Type Systems (IDTSs) [5]. This approach has also been adapted to richer type disciplines like Coquand and Huet's Calculus of Constructions [2, 4], in order to extend the equality used in proof assistants based on the Curry-De Bruijn-Howard isomorphism like Coq [10] or Lego [17].

Although CRSs and HRSs seem quite different, they have been precisely compared by van Oostrom and van Raamsdonk [31], and shown to have the same expressive power, CRSs using a more lazy evaluation strategy than HRSs. On the other hand, although IDTSs seem very close in spirit to CRSs, the relation between both systems has not been clearly stated yet.

Other approaches have been proposed like Wolfram's Higher-Order Term Rewriting Systems (HOTRSs) [33], Khasidashvili's Expression Reduction Systems (ERSs) [14], Takahashi's Conditional Lambda-Calculus (CLC) [27], ... (see [29]). To tame this proliferation, van Oostrom and van Raamsdonk introduced Higher-Order Rewriting Systems (HORSs) [29, 32] in which the matching procedure is a parameter called "substitution calculus". It appears that most of the known approaches can be obtained by using an appropriate substitution calculus. Van Oostrom proved important confluence results for HORSs whose substitution calculus fulfill some conditions, hence factorizing the existing proofs for the different approaches.

Many results have been obtained so far about the confluence of CRSs and HRSs. On the other hand, for IDTSs, termination was the target of research efforts. A powerful and decidable termination criterion has been developed by Jouannaud, Okada and the present author, called the General Schema [5].

So, one may wonder whether the General Schema may be applied to HRSs, and whether Nipkow's higher-order critical pair analysis technique for proving local confluence of HRSs may be applied to IDTSs.

This paper answers positively both questions. However, we do not consider the *critical interpretation* introduced in [5] for dealing with function definitions over strictly positive inductive types (like Brouwer's ordinals or process algebra). In Section 3, we show how IDTSs relate to CRSs and extend IDTSs with the CRS higher-order pattern-matching mechanism, resulting in simply-typed CRSs. In Section 4, we adapt the General Schema to this new calculus and prove in Section 5 that the rewrite systems that follow this schema are strongly normalizing (every reduction sequence is finite). In Section 6, we show that it can be applied to HRSs. In Section 7, we show that Nipkow's higher-order critical pair analysis technique can be applied to IDTSs.

For proving the termination of a HRS, other criteria are available. Van de Pol extended to the higher-order case the use of strictly monotone interpretations [28]. This approach is of course very powerful but it cannot be automated. In [13], Jouannaud and Rubio defined an extension to the higher-order case of Dershowitz' Recursive Path Ordering (HORPO) exploiting the notion of computable closure introduced in [5] by Jouannaud, Okada and the present author for defining the General Schema. Roughly speaking, the General Schema may be seen as a non-recursive version of HORPO. However, HORPO has not yet been adapted to higher-order pattern-matching.

2 Preliminaries

We assume that the reader is familiar with simply-typed λ-calculus [3]. The set $T(\mathcal{B})$ of *types* $s, t, \ldots$ generated from a set $\mathcal{B}$ of *base types* $\mathtt{s}, \mathtt{t}, \ldots$ (in bold font) is the smallest set built from $\mathcal{B}$ and the function type constructor $\rightarrow$. We denote by $FV(u)$ the set of free variables of a term u, $u\downarrow_\beta$ (resp. $u\uparrow^\eta$) the β-normal form of u (resp. the η-long form of u).

We use a postfix notation for the application of substitutions, $\{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\}$ for denoting the substitution θ such that $x_i\theta = u_i$ for each $i \in \{1, \ldots, n\}$, and $\theta \uplus \{x \mapsto u\}$ when $x \notin \mathit{dom}(\theta)$, for denoting the substitution θ' such that $x\theta' = u$ and $y\theta' = y\theta$ if $y \neq x$. The domain of a substitution θ is the set $\mathit{dom}(\theta)$ of variables x such that $x\theta \neq x$. Its codomain is the set $\mathit{cod}(\theta) = \{x\theta \mid x \in \mathit{dom}(\theta)\}$.

Whenever we consider abstraction operators, like $\lambda_{\_.\_}$ in λ-calculus, we work modulo α-conversion, *i.e.* modulo renaming of bound variables. Hence, we can always assume that, in a term, the bound variables are pairwise distinct and distinct from the free variables. In addition, to avoid variable capture when applying a substitution θ to a term u, we can assume that the free variables of the terms of the codomain of θ are distinct from the bound variables of u.

We use words over positive numbers for denoting positions in a term. With a symbol f of fixed arity, say n, the positions of the arguments of f are the numbers $i \in \{1, \ldots, n\}$. We will denote by $Pos(u)$ the set of positions in a term u. The subterm at position p is denoted by $u|_p$. Its replacement by another term v is denoted by $u[v]_p$.

For the sake of simplicity, we will often use vector notations for denoting comma- or space-separated sequences of objects. For example, $\{\vec{x} \mapsto \vec{u}\}$ will denote $\{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\}$, $n = |\vec{u}|$ being the length of $\vec{u}$. Moreover, some functions will be naturally extended to sequences of objects. For example, $FV(\vec{u})$ will denote $\bigcup_{1 \le i \le n} FV(u_i)$ and $\vec{u}\theta$ the sequence $u_1\theta \ldots u_n\theta$.

3 Extending IDTSs with higher-order pattern-matching *à la* CRS

In a Combinatory Rewrite System (CRS) [16], the terms are built from variables $x, y, \dots$ function symbols $f, g, \dots$ of fixed arity and an abstraction operator $[\_]\_$ such that, in $[x]u$, the variable x is bound in u. On the other hand, left-hand and right-hand sides of rules are not only built from variables, function symbols and the abstraction operator like terms, but also from metavariables $Z, Z', \dots$ of fixed arity. In the left-hand sides of rules, the metavariables must be applied to distinct bound variables (a condition similar to the one for patterns *à la* Miller [18]). By convention, a term $Z(x_{i_1}, \dots, x_{i_k})$ headed by $[x_1], \dots, [x_n]$ can be replaced only by a term u such that $FV(u) \cap \{x_1, \dots, x_n\} \subseteq \{x_{i_1}, \dots, x_{i_k}\}$.

For example, in a left-hand side of the form $f([x][y]Z(x))$, the metaterm $Z(x)$ stands for a term in which y cannot occur free, that is, the metaterm $[x][y]Z(x)$ stands for a function of two variables x and y not depending on y.

The λ-calculus itself may be seen as a CRS with the symbol @ of arity 2 for the application, the CRS abstraction operator $[\_]\_$ standing for λ, and the rule

$$@([x]Z(x), Z') \to Z(Z')$$

for the β-rewrite relation. Indeed, by definition of the CRS substitution mechanism, if $Z(x)$ stands for some term u and Z' for some other term v, then $Z(Z')$ stands for $u\{x \mapsto v\}$.

In [5], Inductive Data Type Systems (IDTSs) are defined as extensions of the simply-typed λ-calculus with function symbols of fixed arity defined by rewrite rules. So, an IDTS may be seen as the sub-CRS of well-typed terms, in which the free variables occuring in rewrite rules are metavariables of arity 0, and only β really uses the CRS substitution mechanism.

As a consequence, restricting matching to first-order matching clearly leads to non-confluence. For example, the rule

$$D(\lambda x.sin(F\ x)) \to \lambda x.(D(F)\ x) \times cos(F\ x)$$

defining a formal differential operator D over a function of the form $sin \circ F$, cannot rewrite a term of the form $D(\lambda x.sin(x))$ since x is not of the form $(u\ x)$.

On the other hand, in the CRS approach, thanks to the notions of metavariable and substitution, D may be properly defined with the rule

$$D([x]sin(F(x))) \to [x]\ @(D([y]F(y)), x) \times cos(F(x))$$

where F is a metavariable of arity 1.

This leads us to extend IDTSs with the CRS notions of metavariable and substitution, hence resulting in simply-typed CRSs.

Definition 1 (IDTS - new definition) An *IDTS-alphabet* $\mathcal{A}$ is a 4-tuple $(\mathcal{B}, \mathcal{X}, \mathcal{F}, \mathcal{Z})$ where:

– $\mathcal{B}$ is a set of *base types*,

– $\mathcal{X}$ is a family $(X_t)_{t\in T(\mathcal{B})}$ of sets of *variables*,
– $\mathcal{F}$ is a family $(F_{s_1,\dots,s_n,s})_{n\geq 0,s_1,\dots,s_n,s\in T(\mathcal{B})}$ of sets of *function symbols*,
– $\mathcal{Z}$ is a family $(Z_{s_1,\dots,s_n,s})_{n\geq 0,s_1,\dots,s_n,s\in T(\mathcal{B})}$ of sets of *metavariables*,
such that all the sets are pairwise disjoint.

The set of *IDTS-metaterms* over $\mathcal{A}$ is $\mathcal{I}(\mathcal{A}) = \bigcup_{t\in T(\mathcal{B})} \mathcal{I}_t$ where $\mathcal{I}_t$ are the smallest sets such that:
(1) $X_t \subseteq \mathcal{I}_t$,
(2) if $x \in X_s$ and $u \in \mathcal{I}_t$, then $[x]u \in \mathcal{I}_{s\to t}$,
(3) if $f \in F_{s_1,\dots,s_n,s}$, $u_1 \in \mathcal{I}_{s_1}, \dots, u_n \in \mathcal{I}_{s_n}$, then $f(u_1,\dots,u_n) \in \mathcal{I}_s$.
(4) if $Z \in Z_{s_1,\dots,s_n,s}$, $u_1 \in \mathcal{I}_{s_1}, \dots, u_n \in \mathcal{I}_{s_n}$, then $Z(u_1,\dots,u_n) \in \mathcal{I}_s$.
We say that a metaterm u is *of type* $t \in T(\mathcal{B})$ if $u \in \mathcal{I}_t$. The set of metavariables occuring in a metaterm u is denoted by $Var(u)$. A *term* is a metaterm with no metavariable.

A metaterm l is an *IDTS-pattern* if every metavariable occuring in l is applied to a sequence of distinct bound variables.

An *IDTS-rewrite rule* is a pair $l \to r$ of metaterms such that:
(1) l is an IDTS-pattern,
(2) l is headed by a function symbol,
(3) $Var(r) \subseteq Var(l)$,
(4) r has the same type as l,
(5) l and r are closed $(FV(l) = FV(r) = \emptyset)$.
An n-ary substitute of type $s_1 \to \dots \to s_n \to s$ is an expression of the form $\underline{\lambda}(\vec{x}).u$ where $\vec{x}$ are distinct variables of respective types $s_1,\dots,s_n$ and u is a term of type s. An *IDTS-valuation* σ is a type-preserving map associating an n-ary substitute to each metavariable of arity n. Its (postfix) application to a metaterm returns a term defined as follows:
– $x\sigma = x$
– $([x]u)\sigma = [x]u\sigma \quad (x \notin FV(cod(\sigma)))$
– $f(\vec{u})\sigma = f(\vec{u}\sigma)$
– $Z(\vec{u})\sigma = v\{\vec{x} \mapsto \vec{u}\sigma\}$ if $\sigma(Z) = \underline{\lambda}(\vec{x}).v$
An *IDTS* $\mathcal{I}$ is a pair $(\mathcal{A},\mathcal{R})$ where $\mathcal{A}$ is an IDTS-alphabet and $\mathcal{R}$ is a set of IDTS-rewrite rules over $\mathcal{A}$. Its corresponding rewrite relation $\to_{\mathcal{I}}$ is the subterm compatible closure of the relation containing every pair $l\sigma \to r\sigma$ such that $l \to r \in \mathcal{R}$ and σ is an IDTS-valuation over $\mathcal{A}$.

The following class of IDTSs will interest us especially:

Definition 2 (β-IDTS) An IDTS $(\mathcal{A},\mathcal{R})$ where $\mathcal{A} = (\mathcal{B},\mathcal{X},\mathcal{F},\mathcal{Z})$ is a *β-IDTS* if, for every pair $s,t \in T(\mathcal{B})$, there is:
(1) a function symbol $@_{s,t} \in F_{s\to t,s,t}$,
(2) a rule $\beta_{s,t} = @([x]Z(x), Z') \to Z(Z') \in \mathcal{R}$,
and no other rule has a left-hand side headed by @.

Given an IDTS $\mathcal{I}$, we can always add new symbols and new rules so as to obtain a β-IDTS. We will denote by $\beta\mathcal{I}$ this *β-extension* of $\mathcal{I}$.

For short, we will denote $@(\dots @(@(v,u_1),u_2),\dots,u_n)$ by $@(v,\vec{u})$.

The strong normalization of $\beta\mathcal{I}$ trivially implies the strong normalization of $\mathcal{I}$. However, the study of $\beta\mathcal{I}$ seems a necessary step because the application symbol @ together with the rule β are the essence of the substitution mechanism. Should we replace in the right-hand sides of the rules every metaterm of the form $Z(\vec{u})$ by $@([\vec{x}]Z(\vec{x}), \vec{u})$, the system would lead to the same normal forms.

In Appendix A, we list some results about the relations between $\mathcal{I}$ and $\beta\mathcal{I}$.

4 Definition of the General Schema

All along this section and the following one, we fix a given β-IDTS $\mathcal{I} = (\mathcal{A}, \mathcal{R})$. Firstly, we adapt the definition of the General Schema given in [5] to take into account the notion of metavariable. Then, we prove that if the rules of $\mathcal{R}$ follow this schema, then $\rightarrow_{\mathcal{I}}$ is strongly normalizing.

The General Schema is a syntactic criterion which ensures the strong normalization of IDTSs. It has been designed so as to allow a strong normalization proof by the technique of *computability predicates* introduced by Tait for proving the normalization of the simply-typed λ-calculus [26, 9]. Hereafter, we only give basic definitions. The reader will find more details in [5].

Given a rule with left-hand side $f(\vec{l})$, we inductively define a set of admissible right-hand sides that we call the *computable closure* of $\vec{l}$, starting from the *accessible* metavariables of $\vec{l}$. The main problem will be to prove that the computable closure is indeed a set of "computable" terms whenever the terms in $\vec{l}$ are "computable". This is the objective of Lemma 13 below. The notion of computable closure has been first introduced by Jouannaud, Okada and the present author in [5, 4] for defining the General Schema, but it has been also used by Jouannaud and Rubio in [13] for strengthening their Higher-Order Recursive Path Ordering.

For each base type $\mathtt{s}$, we assume given a set $C_{\mathtt{s}} \subseteq \bigcup_{p \geq 0, s_1, \ldots, s_p \in T(\mathcal{B})} F_{s_1, \ldots, s_p, \mathtt{s}}$ whose elements are called the *constructors* of $\mathtt{s}$. When a function symbol is a constructor, we may denote it by the lower case letters $c, d, \ldots$

This induces the following relation on base types: $\mathtt{t}$ *depends on* $\mathtt{s}$ if there is a constructor $c \in C_{\mathtt{t}}$ such that $\mathtt{s}$ occurs in the type of one of the arguments of c. Its reflexive and transitive closure $\leq_{\mathcal{B}}$ is a quasi-ordering whose associated equivalence relation (resp. strict ordering) will be denoted by $=_{\mathcal{B}}$ (resp. $<_{\mathcal{B}}$).

We say that a constructor $c \in C_{\mathtt{s}}$ is *positive* if every base type $\mathtt{t} =_{\mathcal{B}} \mathtt{s}$ occurs only at positive positions (wrt. the type constructor $\rightarrow$) into the types of the arguments of c. c is *basic* if it is positive and has no functional arguments. A type is *positive* (resp. *basic*) if all its constructors are positive (resp. basic).

Definition 3 (Accessible subterms) The set $Acc(v)$ of *accessible subterms* of a metaterm v is the smallest set such that:

(1) $v \in Acc(v)$

(2) if $[x]u \in Acc(v)$ then $u \in Acc(v)$

(3) if $c(\vec{u}) \in Acc(v)$ then each $u_i \in Acc(v)$

(4) if $f(\vec{u}) \in Acc(v)$ and u_i is of basic type then $u_i \in Acc(v)$
(5) if $@(u, x) \in Acc(v)$, $x \notin FV(u) \cup FV(v)$ then $u \in Acc(v)$
(6) if $@(x, \vec{u}) \in Acc(v)$, $x \notin FV(\vec{u}) \cup FV(v)$ then each $u_i \in Acc(v)$.

By abuse of notation, we will say that a metavariable Z is *accessible* in v if there are distinct bound variables $\vec{x}$ such that $Z(\vec{x}) \in Acc(v)$.

For example, F is accessible in $v = [x]sin(F(x))$ since $sin(F(x))$ is accessible in v by (2), and thus, $F(x)$ is accessible in v by (3).

Compared to [5], we express the accessibility with respect to a fixed v. This has no consequence on the definition of computable closure since, among the accessible subterms, only the free variables (here, the metavariables) are taken into account. Accessibility enjoys the following property:

Property 4 If $u \in Acc(v)$ then $u\sigma \in Acc(v\sigma)$.

For proving termination, we are led to compare the arguments of a function symbol with the arguments of the recursive calls generated by its reductions. To this end, each function symbol $f \in \mathcal{F}$ is equipped with a *status* $stat_f$ which specifies how to make the comparison as a simple combination of multiset and lexicographic comparisons. Then, an ordering on terms $\leq$ is easily extended to an ordering on sequences of terms $\leq_{stat_f}$. The reader will find precise definitions in [5]. To fix an idea, one can assume that $\leq_{stat_f}$ is the lexicographic extension $\leq_{lex}$ or the multiset extension $\leq_{mul}$ of $\leq$. We will denote by $\leq^{>}_{stat_f}$ (resp. $\leq^{\simeq}_{stat_f}$) the strict ordering (resp. equivalence relation) associated to $\leq_{stat_f}$. $\leq^{>}_{stat_f}$ is well-founded if the strict ordering associated to $\leq$ is well-founded.

$\mathcal{R}$ induces the following relation on function symbols: g *depends on* f if there is a rewrite rule defining g (*i.e.* whose left-hand side is headed by g) in the right-hand side of which f occurs. Its reflexive and transitive closure is a quasi-ordering denoted by $\leq_{\mathcal{F}}$ whose associated equivalence relation (resp. strict ordering) will be denoted by $=_{\mathcal{F}}$ (resp. $<_{\mathcal{F}}$).

Finally, we will do the following

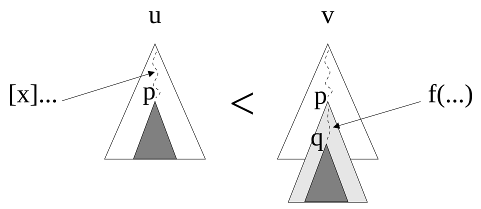
Assumptions (A)
(1) every constructor is positive
(2) no left-hand side of rule is headed by a constructor
(3) both $>_{\mathcal{B}}$ and $>_{\mathcal{F}}$ are well-founded
(4) $stat_f = stat_g$ whenever $f =_{\mathcal{F}} g$

The first assumption comes from the fact that, from non-positive inductive types, it is possible to build non-terminating terms [19]. The second assumption ensures that if a constructor-headed term is computable, then its arguments are computable too. The third assumption ensures that types and function definitions are not cyclic. The fourth assumption says that the arguments of equivalent symbols must be compared in the same way.

For comparing the arguments, the subterm ordering $\trianglelefteq$ used in [5] is not satisfactory anymore because of the metavariables which must be applied to some arguments. For example, $[x]F(x)$ is not a subterm of $[x]sin(F(x))$. This can be repaired by using the following ordering.

Definition 5 (Covered-subterm ordering) We say that a metaterm u is a *covered-subterm* of a metaterm v, written $u \mathrel{\widehat{\trianglelefteq}} v$, if there are two positions $p \in Pos(v)$ and $q \in Pos(v|_p)$ such that (see the figure):

– $u = v[v|_{pq}]_p$,
– $\forall r < p$, $v|_r$ is headed by an abstraction,
– $\forall r < q$, $v|_{pr}$ is headed by a function symbol (which can be a constructor).



Property 6

(1) $\widehat{\rhd}$ is stable by valuation: if $u \mathrel{\widehat{\rhd}} v$ and σ is a valuation, then $u\sigma \mathrel{\widehat{\rhd}} v\sigma$.
(2) $\widehat{\rhd}$ is stable by substitution: if $u \mathrel{\widehat{\rhd}} v$ and θ is a substitution, then $u\theta \mathrel{\widehat{\rhd}} v\theta$.
(3) $\widehat{\rhd}$ commutes with $\rightarrow$: if $u \mathrel{\widehat{\rhd}} v$ and $v \rightarrow w$ then there is a term v' such that $u \rightarrow v'$ and $v' \mathrel{\widehat{\rhd}} w$.

Finally, we come to the definition of computable closure.

Definition 7 (Computable closure) Given a function symbol $f \in F_{s_1,\dots,s_n,s}$, the *computable closure* $\mathcal{CC}_f(\vec{l})$ of a metaterm $f(\vec{l})$ is the least set $\mathcal{CC}$ such that:

(1) if $Z \in Z_{t_1,\dots,t_p,t}$ is accessible in $\vec{l}$ and $\vec{u}$ are p metaterms of $\mathcal{CC}$ of respective types $t_1, \dots, t_p$, then $Z(\vec{u}) \in \mathcal{CC}$;
(2) if $x \in X_t$ then $x \in \mathcal{CC}$;
(3) if $c \in C_{\mathtt{t}} \cap F_{t_1,\dots,t_p,\mathtt{t}}$ and $\vec{u}$ are p metaterms of $\mathcal{CC}$ of respective types $t_1, \dots, t_p$, then $c(\vec{u}) \in \mathcal{CC}$;
(4) if u and v are two metaterms of $\mathcal{CC}$ of respective types $s \rightarrow t$ and s then $@(u, v) \in \mathcal{CC}$;
(5) if $u \in \mathcal{CC}$ then $[x]u \in \mathcal{CC}$;
(6) if $h \in F_{t_1,\dots,t_p,t}$, $h <_{\mathcal{F}} f$ and $\vec{w}$ are p metaterms of $\mathcal{CC}$ of respective types $t_1, \dots, t_p$, then $h(\vec{w}) \in \mathcal{CC}$;
(7) if $g \in F_{t_1,\dots,t_p,t}$, $g =_{\mathcal{F}} f$ and $\vec{u}$ are $p \geq 1$ metaterms of $\mathcal{CC}$ of respective types $t_1, \dots, t_p$ such that $\vec{u} \mathrel{\widehat{\trianglelefteq}^{>}_{stat_f}} \vec{l}$, then $g(\vec{u}) \in \mathcal{CC}$.

Note that we do not consider in case (7) the notion of *critical interpretation* introduced in [5] for proving the termination of function definitions over strictly positive types (like Brouwer's ordinals or process algebra).

Definition 8 (General Schema) A rewrite rule $f(\vec{l}) \rightarrow r$ follows the *General Schema* GS if $r \in \mathcal{CC}_f(\vec{l})$.

A first example is given by the rule β itself: $@([x]Z(x), Z') \to Z(Z')$ (Z and Z' are both accessible).

$D([x]sin(F(x))) \to [x]@(D([y]F(y)), x) \times cos(F(x))$ also follows the General Schema since x and y belong to the computable closure of $[x]sin(F(x))$ by (2), hence $F(x)$ and $F(y)$ by (1) since F is accessible in $[x]sin(F(x))$, $[y]F(y)$ by (5), $D([y]F(y))$ by (7) since $[y]F(y)$ is a strict covered-subterm of $[x]sin(F(x))$, $@(D([y]F(y)), x)$ by (4), $cos(F(x))$ by (3), $@(D([y]F(y)), x) \times cos(F(x))$ by (6) and the whole right-hand side by (5).

5 Termination proof

The termination proof follows Tait's technique of computability predicates [26, 9]. Computability predicates are sets of strongly normalizable terms satisfying appropriate conditions. For each type, we define an interpretation which is a computability predicate and we prove that every term is computable, *i.e.* it belongs to the interpretation of its type. For precise definitions, see [5].

The main things to know are:

– Computability implies strong normalizability.
– If u is a term of type $s \to t$, then it is computable iff, for every computable term v of type s, $@(u, v)$ is computable.
– Computability is preserved by reduction.
– A term is *neutral* if it is neither constructor-headed nor an abstraction. A neutral term u is computable if all its immediate reducts are computable.
– A constructor-headed term $c(\vec{u})$ is computable iff all the terms in $\vec{u}$ are computable.
– For basic types, computability is equivalent to strong normalizability.

Definition 9 (Computable valuation) A substitution is *computable* if all the terms of its codomain are computable. A substitute $\underline{\lambda}(\vec{x}).u$ is *computable* if, for any computable substitution θ such that $dom(\theta) \subseteq \{\vec{x}\}$, $u\theta$ is computable. Finally, a valuation σ is *computable* if, for every metavariable Z, the substitute $\sigma(Z)$ is computable.

Lemma 10 (Compatibility of accessibility with computability) If $u \in Acc(v)$ and v is computable, then for any computable substitution θ such that $dom(\theta) \cap FV(v) = \emptyset$, $u\theta$ is computable.

Proof. By induction on $Acc(v)$. Without loss of generality, we can assume that $dom(\theta) \subseteq FV(u)$ since $u\theta = u\theta|_{FV(u)}$.

(1) Immediate.

(2) θ is of the form $\theta' \uplus \{x \mapsto x\theta\}$ where $dom(\theta') \cap FV(v) = \emptyset$. By induction hypothesis, $([x]u)\theta'$ is computable. By taking x away from $FV(cod(\theta'))$, $([x]u)\theta' = [x]u\theta'$ and $u\theta = u\theta'\{x \mapsto x\theta\}$ is a reduct of $@([x]u\theta', x\theta)$, hence it is computable since $x\theta$ is computable.

(3) By induction hypothesis, $c(\vec{u})\theta = c(\vec{u}\theta)$ is computable. Hence, by definition of the interpretation for inductive types, $u_i\theta$ is computable.

(4) By induction hypothesis, $f(\vec{u})\theta = f(\vec{u}\theta)$ is computable. Hence $u_i\theta$ is strongly normalizable, and since, for terms of basic type, computability is equivalent to strong normalizability, $u_i\theta$ is computable.

(5) u must be of type $s \rightarrow t$. So, let w be a computable term of type s. Since $x \notin FV(u)$, $x \notin dom(\theta)$. Then, let $\theta' = \theta \uplus \{x \mapsto w\}$. θ' is computable and $dom(\theta') \cap FV(v) = \emptyset$ since $x \notin FV(v)$. Hence, by induction hypothesis, $@(u,x)\theta' = @(u\theta, w)$ is computable.

(6) Since $x \notin FV(u)$, $x \notin dom(\theta)$. Then, let $\theta' = \theta \uplus \{x \mapsto [\vec{y}]y_i\}$, $[\vec{y}]y_i$ being the i-th projection. θ' is computable and $dom(\theta') \cap FV(v) = \emptyset$ since $x \notin FV(v)$. Hence, by induction hypothesis, $@(x,\vec{u})\theta' = @([\vec{y}]y_i, \vec{u}\theta)$ is computable and its β-reduct $u_i\theta$ also.

Corollary 11 Let l be a pattern, v a term and σ a valuation such that $l\sigma = v$. If Z is accessible in l and v is computable, then $\sigma(Z)$ is computable.

For proving Lemma 14 below, we will reason by induction on $(f,\vec{u})$ with the ordering $\succeq = (\geq_{\mathcal{F}}, \rightarrow_{mul} \cup \widehat{\trianglerighteq}^{>}_{stat_f})_{lex}$, $\vec{u}$ being strongly normalizable arguments of f. Since $\widehat{\triangleright}$ commutes with $\rightarrow$, we can prove that $\widehat{\trianglerighteq}^{>}_{stat_f}\rightarrow_{mul}$ is included into ${\rightarrow_{mul}}^{0,1}\ \widehat{\trianglerighteq}^{>}_{stat_f}$ where ${\rightarrow_{mul}}^{0,1}$ means zero or one $\rightarrow_{mul}$-step. This implies that $\rightarrow_{mul} \cup \widehat{\trianglerighteq}^{>}_{stat_f}$ is well-founded since:

Lemma 12 If a and b are two well-founded relations such that $ab \subseteq b^*a$ then $a \cup b$ is well-founded.

Therefore the strict ordering $\succ$ associated to $\succeq$ is well-founded since $>_{\mathcal{F}}$ is assumed to be well-founded. Now, we can prove the correctness of the computable closure.

Lemma 13 (Computable closure correctness) Let $f(\vec{l})$ be a pattern. Assume that σ is a computable valuation and that the terms in $\vec{l}\sigma$ are computable. Assume also that, for every function symbol h and sequence of computable terms $\vec{w}$ such that $(f,\vec{l}\sigma) \succ (h,\vec{w})$, $h(\vec{w})$ is computable. Then, for every $r \in \mathcal{CC}_f(\vec{l})$, $r\sigma$ is computable.

Proof. The proof, by induction on $\mathcal{CC}_f(\vec{l})$, is quite similar to the one given in [5] except that, now, one has to deal with valuations instead of substitutions. The main difference is in case (1) for metavariables. We only give this case. A full proof can be found in Appendix C.

In fact, we prove that, for any computable valuation σ such that $FV(cod(\sigma)) \cap FV(r) = \emptyset$, for any computable substitution θ such that $dom(\theta) \subseteq FV(r)$ and for any $r \in \mathcal{CC}_f(\vec{l})$, $r\sigma\theta = r\theta\sigma$ is computable.

(1) $r = Z(\vec{v})$ where Z is a metavariable accessible in $\vec{l}$ and $\vec{v}$ are metaterms of $\mathcal{CC}$. We first prove it for a special case and then for the general case.

(a) $\vec{v}$ is a sequence of distinct bound variables, say $\vec{x}$. Without loss of generality, we can assume that $\sigma(Z) = \underline{\lambda}(\vec{x}).w$. Then, $r\sigma\theta = w\theta$. Since σ is computable and $dom(\theta) \subseteq \{\vec{x}\} = FV(r)$, $w\theta$ is computable.

(b) $r\sigma\theta$ is a β-reduct of the term $@([\vec{x}]Z(\vec{x})\sigma\theta, \vec{v}\sigma\theta)$ where $\vec{x}$ are fresh distinct variables. By case (1a) and (5), $[\vec{x}]Z(\vec{x})\sigma\theta$ is computable and since, by induction hypothesis, the terms in $\vec{v}\sigma\theta$ are also computable, $r\sigma\theta$ is computable.

Lemma 14 (Computability of function symbols) If all the rules satisfy the General Schema then, for every function symbol f, $f(\vec{u})$ is computable whenever the terms in $\vec{u}$ are computable.

Proof. If f is a constructor then this is immediate since the terms in $\vec{u}$ are computable by assumption. Assume now that f is a function symbol. Since $f(\vec{u})$ is neutral, to prove that $f(\vec{u})$ is computable, it suffices to prove that all its immediate reducts are computable. We prove this by induction on $(f, \vec{u})$ with $\succ$ as well-founded ordering.

Let v be an immediate reduct of $f(\vec{u})$. v is either a head-reduct of $f(\vec{u})$ or of the form $f(u_1, \ldots, u_i', \ldots, u_n)$ with u_i' being an immediate reduct of u_i.

In the latter case, as computability predicates are stable by reduction, u_i' is computable. Hence, since $(f, u_1 \ldots u_i' \ldots u_n) \prec (f, \vec{u})$, by induction hypothesis, $f(u_1, \ldots, u_i', \ldots, u_n)$ is computable.

In the former case, there is a rule $f(\vec{l}) \rightarrow r$ and a valuation σ such that $\vec{u} = \vec{l}\sigma$ and $v = r\sigma$. By definition of the computable closure, and since $\mathit{Var}(r) \subseteq \mathit{Var}(\vec{l})$, every metavariable occuring in r is accessible in $\vec{l}$. Hence, since the terms in $\vec{l}\sigma$ are computable, by Corollary 11, $\sigma|_{\mathit{Var}(r)}$ is computable. Therefore, by Lemma 13, $r\sigma = r\sigma|_{\mathit{Var}(r)}$ is computable.

Theorem 15 (Strong normalization) Let $\mathcal{I} = (\mathcal{A}, \mathcal{R})$ be a β-IDTS satisfying the assumptions (A). If all the rules of $\mathcal{R}$ satisfy the General Schema, then $\rightarrow_{\mathcal{I}}$ is strongly normalizing.

Proof. One can easily prove that, for every term u and computable substitution θ, $u\theta$ is computable. In case where $u = f(\vec{u})$, we conclude by Lemma 14. The theorem follows easily since the identity substitution is computable.

It is possible to improve this termination result as follows. After [12], if $\mathcal{R}$ follows the General Schema and $\mathcal{R}_1$ is a terminating set of non-duplicating[1] first-order rewrite rules, then $\mathcal{R} \cup \mathcal{R}_1$ is also terminating.

6 Application of the General Schema to HRSs

We just recall what is a HRS. The reader can find precise definitions in [18]. A HRS $\mathcal{H}$ is a pair $(\mathcal{A}, \mathcal{R})$ made of a HRS-alphabet $\mathcal{A}$ and a set $\mathcal{R}$ of HRS-rewrite rules over $\mathcal{A}$. A HRS-alphabet is a triple $(\mathcal{B}, \mathcal{X}, \mathcal{F})$ where $\mathcal{B}$ is a set of base types, $\mathcal{X}$ is a family $(X_s)_{s \in T(\mathcal{B})}$ of variables and $\mathcal{F}$ is a family $(F_s)_{s \in T(\mathcal{B})}$ of function

[1] No metavariable occurs more often in the right-hand side than in the left-hand side.

symbols. The corresponding HRS-terms are the terms of the simply-typed λ-calculus built over $\mathcal{X}$ and $\mathcal{F}$ that are in η-long β-normal form.

So, a HRS $\mathcal{H}$ can be seen as an IDTS $\langle\mathcal{H}\rangle$ with the same symbols, the arity of which being determined by the maximum number of arguments they can take, plus the symbol @ for the application. Hence it is a β-IDTS. In [31], van Oostrom and van Raamsdonk studied this translation in detail and proved:

Lemma 16 (Van Oostrom and van Raamsdonk [31]) Let $\mathcal{H}$ be a HRS. If $u \rightarrow_{\mathcal{H}} v$ then $\mathcal{I}(u) \rightarrow_{\mathcal{I}(\mathcal{H})}\rightarrow^*_\beta \mathcal{I}(v)$ where $\mathcal{I}(v)$ is in β-normal form.

As a consequence, $\mathcal{H}$ is strongly normalizing if $\langle\mathcal{H}\rangle$ so is. Thus, the General Schema can be used on $\langle\mathcal{H}\rangle$ for proving the termination of $\mathcal{H}$. In fact, it can be used directly on $\mathcal{H}$ if we adapt the notions of accessible subterm and computable closure to HRSs. See Appendix B for details.

Theorem 17 (Strong normalization for HRSs) Let $\mathcal{H} = (\mathcal{A}, \mathcal{R})$ be a HRS satisfying the assumptions (A). If all the rules of $\mathcal{R}$ satisfy the General Schema for HRSs, then $\rightarrow_{\mathcal{H}}$ is strongly normalizing.

Proof. This results from the fact proved in Appendix B that, if $\mathcal{H}$ follows the General Schema for HRSs then $\langle\mathcal{H}\rangle$ follows the General Schema for IDTSs.

7 Confluence of IDTSs

First of all, since an IDTS is a sub-CRS, it is confluent whenever the underlying CRS is confluent. This is the case if it is weakly orthogonal, *i.e.* it is left-linear and all (higher-order) critical pairs are equal [29], or if it is left-linear and all critical pairs are development closed [30].

Now, one may wonder whether Nipkow's result for local confluence of HRSs [18] may be applied to IDTSs. To this end, we need to interpret an IDTS as a HRS. This can be done in the following natural way:

Definition 18 (Natural translation of IDTSs into HRSs) An IDTS-alphabet $\mathcal{A} = (\mathcal{B}, \mathcal{X}, \mathcal{F}, \mathcal{Z})$ can be naturally translated into the HRS-alphabet $\mathcal{H}(\mathcal{A}) = (\mathcal{B}, \mathcal{X}', \mathcal{F}')$ where:

– $X'_{s_1\rightarrow\ldots\rightarrow s_n\rightarrow \mathsf{s}} = X_{s_1\rightarrow\ldots\rightarrow s_n\rightarrow \mathsf{s}} \cup \bigcup_{0\le p\le n} Z_{s_1,\ldots,s_p,s_{p+1}\rightarrow\ldots\rightarrow s_n\rightarrow \mathsf{s}}$

– $F'_{s_1\rightarrow\ldots\rightarrow s_n\rightarrow \mathsf{s}} = \bigcup_{0\le p\le n} F_{s_1,\ldots,s_p,s_{p+1}\rightarrow\ldots\rightarrow s_n\rightarrow \mathsf{s}}$

An IDTS-metaterm u is naturally translated into a HRS-term $\mathcal{H}(u)$ as follows:

– $\mathcal{H}(x) = x\!\uparrow^\eta$ – $\mathcal{H}(f(\vec{u})) = (f\ \mathcal{H}(\vec{u}))\!\uparrow^\eta$

– $\mathcal{H}([x]u) = \lambda x.\mathcal{H}(u)$ – $\mathcal{H}(Z(\vec{u})) = (Z\ \mathcal{H}(\vec{u}))\!\uparrow^\eta$

Finally, an IDTS $\mathcal{I} = (\mathcal{A}, \mathcal{R})$ is translated into the HRS $\mathcal{H}(\mathcal{I}) = (\mathcal{H}(\mathcal{A}), \mathcal{H}(\mathcal{R}))$ where $\mathcal{H}(\mathcal{R}) = \{\mathcal{H}(l) \rightarrow \mathcal{H}(r) \mid l \rightarrow r \in \mathcal{R}\}$.

However, for Nipkow's result to hold, the rewrite rules must be of base type, which is not necessarily the case for IDTSs. This is why, in their study of the relations between CRSs and HRSs [31], van Oostrom and van Raamsdonk defined a translation from CRSs to HRSs, also denoted by $\langle\ \rangle$, which uses a new symbol Λ for forcing the translated terms to be of base type. Furthermore, they proved that (1) if $u \rightarrow_{\mathcal{I}} v$ then $\langle u\rangle \rightarrow_{\langle\mathcal{I}\rangle} \langle v\rangle$, and (2) if $\langle u\rangle \rightarrow_{\langle\mathcal{I}\rangle} v'$ then there is a term v such that $\langle v\rangle = v'$ and $u \rightarrow_{\mathcal{I}} v$. In fact, it is no more difficult to prove the same property for the translation $\mathcal{H}$. As a consequence, since $\langle\ \rangle$ (resp. $\mathcal{H}$) is injective, the (local) confluence of $\langle\mathcal{I}\rangle$ (resp. $\mathcal{H}(\mathcal{I})$) implies the (local) confluence of $\mathcal{I}$. Thus it is possible to deduce the local confluence of $\mathcal{I}$ from the analysis of the critical pairs of $\langle\mathcal{I}\rangle$ (resp. $\mathcal{H}(\mathcal{I})$), and indeed, it turns out that $\langle\mathcal{I}\rangle$ and $\mathcal{H}(\mathcal{I})$ have the "same" critical pairs (see the proof of Theorem 19 in Appendix C for details). Identifying $\mathcal{I}$ with its natural translation $\mathcal{H}(\mathcal{I})$, we claim that:

Theorem 19 If every critical pair of $\mathcal{I}$ is confluent, then $\mathcal{I}$ is locally confluent.

It could also have been possible to consider the translation $\mathcal{H}'$ which is identical to $\mathcal{H}$ but pulls down to base type the rewrite rules by taking $\mathcal{H}'(f(\vec{l}) \rightarrow r) = (f\ \mathcal{H}(\vec{l})\ \vec{x}) \rightarrow v$ if $\mathcal{H}(r) = \lambda\vec{x}.v$ with v of base type. Note that the left-hand side is still a pattern. Then, it is possible to prove that $\mathcal{H}(\mathcal{I})$ and $\mathcal{H}'(\mathcal{I})$ have also the same critical pairs.

8 Conclusion

In Inductive Data Type Systems (IDTSs) [5], the use of first-order matching does not allow to define some functions as expected, resulting in non-confluent computations. By extending IDTS with the higher-order pattern-matching mechanism of Klop's Combinatory Reduction Systems (CRSs) [16], we solved this problem and made clear the relation between IDTSs and CRSs: IDTSs with higher-order pattern-matching are simply-typed CRSs.

We extended a decidable termination criterion defined for IDTSs with first-order matching and called the General Schema [5] to the case of higher-order pattern-matching, and we proved that a rewrite system following this schema is strongly-normalizing.

We also compared this unified approach to Nipkow's Higher-order Rewrite Systems (HRSs) [18]. First, we proved that the extended General Schema can be applied to HRSs. Second, we show how Nipkow's higher-order critical pair analysis technique for proving local confluence can be applied to IDTSs.

Now, several extensions should be considered.

We did not take into account the interpretation defined in [5] for dealing with definitions over strictly positive types (like Brouwer's ordinals or process algebra). However, we expect that it can also be adapted to higher-order pattern-matching.

It is also important to be able to relax the pattern condition which says that metavariables must be applied to distinct bound variables. But it is not clear how to prove the termination with Tait's computability predicates technique when this condition is not satisfied.

Another point is that some computations often need to be performed within some equational theories like commutativity or commutativity and associativity of some function symbols. It would be interesting to know if the General Schema technique can be adapted for dealing with such equational theories.

Finally, one may wonder whether all these results could be establish in the more general framework of van Oostrom and van Raamsdonk's Higher-Order Rewriting Systems (HORSs) [29, 32], under some suitable conditions over the substitution calculus.

Acknowledgments: I am very grateful to D. Kesner, A. Boudet and J.-P. Jouannaud for their suggestions and remarks on previous versions of this paper. I also thank the anonymous referees for their useful comments.

References

[1] P. Aczel. A general Church-Rosser theorem. Technical report, University of Manchester, United Kingdom, 1978.

[2] F. Barbanera, M. Fernández, and H. Geuvers. Modularity of strong normalization in the algebraic-λ-cube. *Journal of Functional Programming*, 7(6), 1997.

[3] H. Barendregt. Lambda calculi with types. In S. Abramski, D. M. Gabbai, and T. S. E. Maiboum, editors, *Handbook of logic in computer science*, volume 2. Oxford University Press, 1992.

[4] F. Blanqui, J.-P. Jouannaud, and M. Okada. The Calculus of Algebraic Constructions. In *Proc. of RTA'99, LNCS 1631.*

[5] F. Blanqui, J.-P. Jouannaud, and M. Okada. Inductive Data Type Systems, 1998. To appear in TCS. Available at `http://www.lri.fr/~blanqui/`.

[6] V. Breazu-Tannen. Combining algebra and higher-order types. In *Proc. of LICS'88, IEEE Computer Society.*

[7] V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic strong normalization. In *Proc. of ICALP'89, LNCS 372.*

[8] V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic strong normalization. *Theoretical Computer Science*, 83(1), 1991.

[9] J. R. Hindley and J. P. Seldin. *Introduction to combinators and λ-calculus.* London Mathematical Society, 1986.

[10] INRIA-Rocquencourt/CNRS/Université Paris-Sud/ENS Lyon, France. *The Coq Proof Assistant Reference Manual Version 6.3*, 1999. Available at `http://pauillac.inria.fr/coq/`.

[11] J.-P. Jouannaud and M. Okada. Executable higher-order algebraic specification languages. In *Proc. of LICS'91, IEEE Computer Society.*

[12] J.-P. Jouannaud and M. Okada. Abstract Data Type Systems. *Theoretical Computer Science*, 173(2), 1997.

[13] J.-P. Jouannaud and A. Rubio. The Higher-Order Recursive Path Ordering. In *Proc. of LICS'99, IEEE Computer Society.*

[14] Z. Khasidashvili. Expression Reduction Systems. In *Proc. of I. Vekua Institute of Applied Mathematics*, volume 36, 1990.

[15] J. W. Klop. *Combinatory Reduction Systems.* PhD thesis, University of Utrecht, Netherlands, 1980. Published as Mathematical Center Tract 129.

[16] J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121(1-2), 1993.

[17] Z. Luo and R. Pollack. *LEGO Proof Development System: User's manual.* University of Edinburgh, Scotland, 1992.

[18] R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192, 1998.

[19] N. P. Mendler. *Inductive Definition in Type Theory.* PhD thesis, Cornell University, United States, 1987.

[20] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In *Proc. of ELP'89, LNCS 475.*

[21] D. Miller and G. Nadathur. An overview of λProlog. In *Proc. of the 5th Int. Conf. on Logic Programming, 1988.*

[22] F. Müller. Confluence of the lambda calculus with left-linear algebraic rewriting. *Information Processing Letters*, 41, 1992.

[23] T. Nipkow. Higher-order critical pairs. In *Proc. of LICS'91, IEEE Computer Society.*

[24] M. Okada. Strong normalizability for the combined system of the typed lambda calculus and an arbitrary convergent term rewrite system. In *Proc. of ISSAC'89, ACM Press.*

[25] L. Paulson. Isabelle: a generic theorem prover. *LNCS 828*, 1994.

[26] W. W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2), 1967.

[27] M. Takahashi. λ-calculi with conditional rules. In *Proc. of TLCA'93, LNCS 664.*

[28] J. van de Pol and H. Schwichtenberg. Strict functionals for termination proofs. In *Proc. of TLCA'95, LNCS 902.*

[29] V. van Oostrom. *Confluence for Abstract and Higher-Order Rewriting.* PhD thesis, Vrije Universiteit, Netherlands, 1994.

[30] V. van Oostrom. Development closed critical pairs. In *Proc. of HOA'95, LNCS 1074*, 1995.

[31] V. van Oostrom and F. van Raamsdonk. Comparing Combinatory Reduction Systems and Higher-order Rewrite Systems. In *Proc. of HOA'93, LNCS 816.*

[32] F. van Raamsdonk. *Confluence and Normalization for Higher-Order Rewriting.* PhD thesis, Vrije Universiteit, Netherlands, 1996.

[33] D. Wolfram. *The clausal theory of types.* PhD thesis, University of Cambridge, United Kingdom, 1990.