Inductive-data-type Systems

Frédéric Blanqui, Jean-Pierre Jouannaud, Mitsuhiro Okada

Inductive Data Type Systems

Frédéric Blanqui, Jean-Pierre Jouannaud

LRI, Bât. 490, Université Paris-Sud
91405 Orsay, FRANCE
Tel: (33) 1.69.15.69.05 Fax: (33) 1.69.15.65.86
`http://www.lri.fr/~blanqui/`

Mitsuhiro Okada

Department of Philosophy, Keio University,
108 Minatoku, Tokyo, JAPAN

Abstract

In a previous work ("Abstract Data Type Systems", TCS 173(2), 1997), the last two authors presented a combined language made of a (strongly normalizing) algebraic rewrite system and a typed l-calculus enriched by pattern-matching definitions following a certain format, called the "General Schema", which generalizes the usual recursor definitions for natural numbers and similar "basic inductive types". This combined language was shown to be strongly normalizing. The purpose of this paper is to reformulate and extend the General Schema in order to make it easily extensible, to capture a more general class of inductive types, called "strictly positive", and to ease the strong normalization proof of the resulting system. This result provides a computation model for the combination of an algebraic specification language based on abstract data types and of a strongly typed functional language with strictly positive inductive types.

Key words: Higher-order rewriting. Strong normalization. Inductive types. Recursive definitions. Typed lambda-calculus.

1 Introduction

This work is one step in a long term program aiming at building formal specification languages integrating computations and proofs within a single framework. We focus here on incorporating an expressive notion of equality within a typed l-calculus.

In retrospect, the quest for an expressive language allowing to specify and prove mathematical properties of software started with system F on the one hand [23,24] and the Automath project on the other hand [15]. Much later, Coquand and Huet combined both calculi, resulting in the Calculus of Constructions [13]. Making use of impredicativity, data structures could be encoded in this calculus, but these encodings were far too complex to be used by non-specialists. A different approach was taken by Martin-Löf [32,33], whose theory was based on the notion of inductive definition, originating in Gödel's system T [25]. Coquand and Paulin-Möhring later incorporated a similar notion to the Calculus of Constructions under the name of inductive type [14]. But despite their legitimate success, inductive types are not yet enough to make the Calculus of Inductive Constructions an easy to use programming language for proofs. The main remaining problem is that of equality. In the current version of the calculus, equality is given by $\beta\eta$-reductions, the recursor rules associated with the inductive types –corresponding to structural induction in the Curry-Howard isomorphism–, and the definitional rules for constants by primitive recursion of higher type. This notion of equality has two main practical drawbacks: it makes the definition of functions sometimes painful for the user, by forcing the user to think operationally rather than axiomatically; it makes it necessary to spell out many equational proofs that could be short-cutted if the corresponding equality could be equationally specified in the calculus.

It should be clear that this problem is not specific to the Calculus of Inductive Constructions. It also shows up in other versions of type theory where equality is not a first-class concept, for example, in Martin Lf's theory of types. A solution was proposed by Coquand, for a calculus with dependent types, in which functions can be defined by pattern-matching, provided all right-hand side recursive calls are "structurally smaller" than the left-hand side call [12]. His notion is very abstract, though, and relies on a well-foundedness assumption which is satisfied in practice. Concurrently, following the pioneering works of Tannen [8], Tannen and Gallier [9,10] and Okada [40], the last two authors of the present paper proposed another solution, for a polymorphically typed l-calculus, based on pattern-matching functional definitions following the so-called "General Schema" [27,28]. This work was then generalized so as to cover the full Calculus of Constructions [1,2,3]. As in Coquand [12], the idea of the General Schema is to control the arguments of the right-hand side recursive calls of a rule-based definition by checking that they are smaller than the left-hand sides ones, this time in the strict subterm ordering extended in a multiset or lexicographic manner. This schema was general enough to subsume basic inductive types, such as $\mathtt{nat} = 0_{\mathtt{nat}} \uplus s_{\mathtt{nat}}(\mathtt{nat})$, in the sense that the associated recursor rules are instances of the General Schema. In contrast with Coquand's proposal, it does not subsume non-basic inductive types, such as $\mathtt{ord} = 0_{\mathtt{ord}} \uplus s_{\mathtt{ord}}(\mathtt{ord}) \uplus lim(\mathtt{nat} \to \mathtt{ord})$, whose constructor lim takes an argument of the functional type $\mathtt{nat} \to \mathtt{ord}$. On the other hand, the use of multiset and lexicographic extensions allows to tailor the comparisons to the

practical needs, making it possible to have nested recursive calls, an important facility that Coquand's ordering cannot provide with. Finally, it is important to note that, in contrast with other work [35,20], our definitions allow non-linear and overlapping left-hand sides, to the price of checking confluence via the computation of critical pairs.

The fact that the General Schema covers only a limited portion of the possible inductive types of the Calculus of Inductive Constructions shows a weakness, and indeed, functions defined by induction over such inductive types cannot be defined within the schema. The purpose of this paper is to revisit the General Schema so as to cover all strictly positive inductive types. The solution is based on an essential use of the positivity condition required for the inductive types. We do so within the framework of Church's simple theory of types, therefore avoiding the problem of having equalities at the type level via the use of dependent types. Closing the gap between the simple theory of types and the Calculus of Inductive Constructions will require further generalizations of the General Schema allowing for dependent and polymorphic inductive types.

The strong normalization proof of our new calculus is based on Tait's computability predicates method [46,24]. In contrast with [28], the whole structure of the proof is made quite modular thanks to a novel formulation of our new version of the General Schema. Here, given a left-hand side $f(\vec{l})$, we define the (infinite) set of possible right-hand sides r such that the rule $f(\vec{l}) \to r$ follows the schema. This set of right-hand sides is generated inductively from $\vec{l}$ by computability preserving operations. This new definition, as it can be easily seen, is strictly stronger than the previous one, allows to reason by induction on the construction of the set of possible right-hand sides, and is easily extensible. This latter feature should prove very useful when extending the present work to the Calculus of Inductive Constructions.

We define our language in Section 2, ending with the new definition of the General Schema in Subsection 2.3. The normalization proof is given in Section 3. In Section 4, we detail many examples and explain possible extension of the General Schema in order to be able to prove some of them. We conclude in Section 5 with two more, important open problems.

2 Inductive Data Type Systems

Intuitively, an *Inductive Data Type System* (IDTS) is a simply-typed l-calculus in which each base type is equipped with a set of constructors together with the associated structural induction principle in the form of Gödel's primitive recursive rules of higher type and additional function symbols (completely) defined by appropriate higher-order rewrite rules. The former kind of rules

can actually be seen as a particular case of the latter, resulting in a uniform formalism with a strong rewriting flavor. In the sequel, we assume the reader familiar with the notions of l-calculus and term rewriting, as presented in [4] for the simply-typed l-calculus, [16] for term rewriting and [31,39,49] for the several variants of higher-order rewriting existing in the literature.

We first introduce the term language before to move on with the definition of higher-order rewrite rules and of the new formulation of the General Schema.

2.1 The language

In this subsection, we introduce successively the signature (made of inductive types, constructors and function symbols) and the set of well-formed terms before to end up with the set of computational rules.

2.1.1 Signature

Definition 1 (Types) *Given a set $\mathcal{I}$ whose elements are called* inductive types*, the set $\mathcal{T}$ of* types *is generated by the following grammar rule:*

$$s = \mathtt{s} \mid (s \to s)$$

where `s` *ranges over $\mathcal{I}$. Furthermore, we consider that $\to$ associates to the right, hence $s_1 \to (s_2 \to s_3)$ can be written $s_1 \to s_2 \to s_3$.*

The sets of positive and negative positions of a type s are inductively defined as follows:

$$\begin{array}{rcl} Pos^+(s \in \mathcal{I}) & = & \epsilon \\ Pos^-(s \in \mathcal{I}) & = & \emptyset \\ Pos^+(s \to t) & = & 1{\cdot}Pos^-(s) \cup 2{\cdot}Pos^+(t) \\ Pos^-(s \to t) & = & 1{\cdot}Pos^+(s) \cup 2{\cdot}Pos^-(t) \end{array}$$

We say that an inductive type `t` *occurs* positively *in a type s if* `t` *does occur in s and every occurrence of* `t` *in s belongs to $Pos^+(s)$.* `t` *is said to occur* strictly positively *in $s_1 \to \ldots \to s_n \to$* `t` *if* `t` *occurs in no s_i.*

This notion of positivity/negativity associated to the type constructor $\to$ is similar to the one used in logic with respect to the implication operator $\Rightarrow$ (as can be expected from the Curry-Howard isomorphism). Note that if `s` does not occur positively in t then, either `s` does not occur in t or else `s` occurs at a negative position in t. For example, `ord` occurs positively in $s =$ `nat` $\to$ `ord` since it occurs in s at the set of positive positions $\{1\} \subseteq Pos^+(s) = \{1\}$. In

fact, it does occur strictly positively since `ord` does not occur in `nat`. On the other hand, `ord` does not occur positively in $t = \texttt{ord} \to \texttt{ord}$ since it occurs at the negative position $1 \in Pos^-(t) = \{1\}$.

Definition 2 (Constructors) *We assume that each inductive type* $\texttt{s} \in \mathcal{I}$ *comes along with an associated set* $\mathcal{C}(\texttt{s})$ *of* constructors*, each constructor* $C \in \mathcal{C}(\texttt{s})$ *being equipped with a type* $\tau(C) = s_1 \to \ldots \to s_n \to \texttt{s}$*. n is called the* arity *of C and we denote by* $\mathcal{C}^n$ *the set of constructors of arity n. We assume that the sets* $\mathcal{C}(\texttt{s})$ *are pairwise disjoint.*

Constructor declarations define a quasi-ordering on $\mathcal{I}$*: an inductive type* `s` *depends on an inductive type* `t`*, written* $\texttt{s} \geq_{\mathcal{I}} \texttt{t}$*, if* `t` *occurs in the type of a constructor* $C \in \mathcal{C}(\texttt{s})$*. (In fact, we consider the reflexive and transitive closure of this relation.) We use* $=_{\mathcal{I}}$ *and* $>_{\mathcal{I}}$ *for respectively the equivalence and the strict ordering associated to* $\geq_{\mathcal{I}}$ *and say that* `s` *is* $\mathcal{I}$-equivalent *to* `t` *if* $\texttt{s} =_{\mathcal{I}} \texttt{t}$*.*

Definition 3 (Strictly positive inductive types) *An inductive type* `s` *is said to be* strictly positive *if it does not occur or occurs strictly positively in the types of the arguments of its constructors, and no type* $\mathcal{I}$*-equivalent to* `s` *occurs at a negative position in the types of the arguments of the constructors of* `s`*. A strictly positive type is* basic *if its constructors have no functional arguments.*

Assumption 1: We assume that $>_{\mathcal{I}}$ is well-founded and that all inductive types are strictly positive.

To spell out the strict-positivity condition, assume that an inductive type `s` has n constructors $C_1, \ldots, C_n$ with $\tau(C_i) = s_{i,1} \to \ldots \to s_{i,n_i} \to \texttt{s}$ and $s_{i,j} = s_{i,j,1} \to \ldots \to s_{i,j,n_{i,j}} \to \texttt{t}_{i,j}$. Then, `s` is strictly positive if $\texttt{t}_{i,j} \leq_{\mathcal{I}} \texttt{s}$, `s` occurs in no $s_{i,j,k}$ and no type $\mathcal{I}$-equivalent to `s` occurs at a negative position in some $s_{i,j}$. It is basic if, moreover, $n_{i,j} = 0$ for all i, j.

Examples of type definitions used in the paper are `bool` for booleans, `nat` for natural numbers, `list_nat` for lists of natural numbers (we do not consider polymorphic types here), `tree` and `list_tree` for the mutually inductive types of trees and lists of trees, `proc` for process expressions [44] (δ denotes the deadlock, ";" the sequencing, + the choice operator and Σ the dependent choice), `ord` for well-founded trees, i.e. Brouwer's ordinals [45], `form` for formulas of the predicate calculus and `R` for expressions built upon real numbers [42]:

- $\texttt{bool} = \mathit{true} : \texttt{bool} \mid \mathit{false} : \texttt{bool}$
- $\texttt{nat} = 0 : \texttt{nat} \mid s : \texttt{nat} \to \texttt{nat}$
- $\texttt{listnat} = \mathit{nil} : \texttt{listnat} \mid \mathit{cons} : \texttt{nat} \to \texttt{listnat} \to \texttt{listnat}$
- $\texttt{tree} = \mathit{node} : \texttt{listtree} \to \texttt{tree}$
- $\texttt{listtree} = \mathit{nil} : \texttt{listtree} \mid \mathit{cons} : \texttt{tree} \to \texttt{listtree} \to \texttt{listtree}$

- $\mathtt{proc} = \delta : \mathtt{proc} \mid ; : \mathtt{proc} \to \mathtt{proc} \to \mathtt{proc} \mid + : \mathtt{proc} \to \mathtt{proc} \to \mathtt{proc} \mid \Sigma : (\mathtt{data} \to \mathtt{proc}) \to \mathtt{proc}$
- $\mathtt{ord} = 0 : \mathtt{ord} \mid s : \mathtt{ord} \mid lim : (\mathtt{nat} \to \mathtt{ord}) \to \mathtt{ord}$
- $\mathtt{form} = \vee : \mathtt{form} \to \mathtt{form} \to \mathtt{form} \mid \neg : \mathtt{form} \to \mathtt{form} \mid \forall : (\mathtt{term} \to \mathtt{form}) \to \mathtt{form} \mid \ldots$
- $\mathtt{R} = 0 : \mathtt{R} \mid 1 : \mathtt{R} \mid + : \mathtt{R} \to \mathtt{R} \to \mathtt{R} \mid cos : \mathtt{R} \to \mathtt{R} \mid ln : \mathtt{R} \to \mathtt{R} \mid \ldots$

All types above are basic, except `ord` and `form` which are strictly positive. We have used the same name for constructors of different types, but we should not if they have to live together. For the sake of simplicity, we will continue in practice to overload names when there is no ambiguity, otherwise we will disambiguate names as in $0_{\mathtt{nat}}$. Our inductive types above are inhabited by expressions built up from their constructors, as for example $\forall(lx.(P\ x) \wedge (Q\ x))$ which represents the logical formula $\forall x\, P(x) \wedge Q(x)$.

A more general class of inductive types is the one of *positive* inductive types. An inductive type is said to be positive if it occurs only at positive positions in the types of the arguments of its constructors (the case of mutually inductive types is defined similarly, by requiring that any type equivalent to it occurs only at positive positions in the types of the arguments of its constructors). The positivity condition ensures that we can define sets of objects by induction on the structure of the elements of the inductive type: it implies the monotonicity of the functional of which the set of objects is the least fixpoint. The class of positive inductive types is the largest class that one can consider within the framework of the simply-typed l-calculus, since any non-positive type is inhabited by non-terminating well-typed terms in this framework [36]. In this paper, we restrict ourselves to strictly positive inductive types, as in the Calculus of Inductive Constructions [51], and prove the strong normalization property of our calculus under this assumption. However, we conjecture that strong normalization holds in the non-strictly positive case too.

Definition 4 (Function symbols) *For each non empty sequence $s_1, \ldots, s_n$, s of types, we assume given a set $\mathcal{F}_{s_1,\ldots,s_n,s}$ of* function symbols *containing the constructors of arity n and type $s_1 \to \ldots \to s_n \to s$. Given a symbol $f \in \mathcal{F}_{s_1,\ldots,s_n,s}$, n is its* arity *and $\tau(f) = s_1 \to \ldots \to s_n \to s$ its* type*. We denote by $\mathcal{F}^n$ the set of function symbols of arity n and by $\mathcal{F}$ the set of all function symbols.*

We also assume given a quasi-ordering $\geq_{\mathcal{F}}$ on $\mathcal{F}$, called precedence*, whose associated strict ordering $>_{\mathcal{F}}$ is well-founded.*

For example, we may have an injection function i from `nat` to `ord`. Then, $lim(ln.i(n))$ represents the first limit ordinal ω as the limit of the infinite sequence of ordinals $0, s(0), s(s(0)), \ldots$ We will later see how to define this injection function in our calculus.

2.1.2 Terms

Definition 5 (Terms) *Given a family* $(\mathcal{X}^s)_{s\in\mathcal{T}}$ *of disjoint infinite sets of* variables *with* $\mathcal{X}$ *denoting their union, the set of* untyped terms *is defined by the grammar rule:*

$$u = x \mid lx.u \mid (u\ u) \mid f(u_1, \ldots, u_n)$$

where f *ranges over* $\mathcal{F}^n$ *and* x *over* $\mathcal{X}$*.* $lx.u$ *denotes the* abstraction *of* u *w.r.t.* x*, i.e. the function of parameter* x *and body* u*, while* $(u\ v)$ *denotes the* application *of the function* u *to the term* v*. A term of the form* $f(u_1, \ldots, u_n)$ *is said to be* function-headed *and* constructor-headed *if* $f \in \mathcal{C}$*.*

The family of sets $(\L^s)_{s\in\mathcal{T}}$ *of* terms of type s *is inductively defined on the structure of terms as follows:*

- *if* $x \in \mathcal{X}^s$ *then* $x \in \L^s$*,*
- *if* $x \in \L^s$ *and* $u \in \L^t$ *then* $lx.u \in \L^{s\to t}$*,*
- *if* $u \in \L^{s\to t}$ *and* $v \in \L^s$ *then* $(u\ v) \in \L^t$*,*
- *if* f *is a function symbol of arity* n *and type* $s_1 \to \ldots \to s_n \to s$ *and* $u_1 \in \L^{s_1}, \ldots, u_n \in \L^{s_n}$ *then* $f(u_1, \ldots, u_n) \in \L^s$*.*

Finally, we denote by $\L = \bigcup_{s\in\mathcal{T}} \L_s$ *the set of* terms *of our calculus. The* type of a term u *is the (unique) type* $t \in \mathcal{T}$ *such that* $u \in \L^t$*. We may use the notation* $u : t$ *to indicate that* u *is of type* t*.*

Note that we could have adopted a presentation based on type-checking rules. The reader will easily extract such rules from the definition of the sets Ł^s.

As usual, we consider that the application associates to the left such that $((u_1\ u_2)\ u_3)$ can be written $(u_1\ u_2\ u_3)$. The sequence of terms $u_1 \ldots u_n$ is denoted by the vector $\vec{u}$ of length $|\vec{u}| = n$. We consider that $(v\ \vec{u})$ and $l\vec{x}.v$ both denote the term v if $\vec{u}$ or $\vec{x}$ is the empty sequence, and the respective terms $(\ldots((v\ u_1)\ u_2)\ldots u_n)$ and $lx_1 \ldots lx_n.v$ otherwise.

After Dewey, the set $Pos(u)$ of *positions* in a term u is a language over the alphabet of strictly positive natural numbers. The *subterm* of a term u at position $p \in Pos(u)$ is denoted by $u|_p$ and the term obtained by replacing $u|_p$ by a term v is written $u[v]_p$. We write $u \trianglerighteq v$ if v is a subterm of u.

We denote by $FV(u)$ the set of *free variables* occurring in a term u. A term in which a variable x occurs freely at most once is said to be *linear w.r.t.* x, and a term is *linear* if all its free variables are linear.

A *substitution* θ is an application from $\mathcal{X}$ to Ł, written in a postfix notation as in $x\theta$. Its *domain* is the set $dom(\theta) = \{x \in \mathcal{X} \mid x\theta \neq x\}$. A substitution

is naturally extended to an application from Ł to Ł, by replacing each free variable by its image and avoiding variable captures. This can be carried out by renaming the bound variables if necessary, an operation called *α-conversion*. As usual, we will always work modulo α-conversion, hence identifying the terms that only differ from each other in their bound variables. Furthermore, we will always assume that free and bound variables are distinct and that bound variables are distinct from each other. Finally, we may use the notation $\{\vec{x} \mapsto \vec{u}\}$ for denoting the substitution which associates u_i to x_i for each i.

2.1.3 Computational rules

Our language is made of three ingredients: a typed l-calculus, a set of inductive types with their constructors and a set of function symbols. As a consequence, there will be three kinds of rules in the calculus: the two rules coming from the l-calculus,

$$\begin{array}{lll} (lx.u\ v) & \rightarrow_\beta & u\{x \mapsto v\} \\ lx.(u\ x) & \rightarrow_\eta & u \qquad\qquad \text{if } x \notin FV(u) \end{array}$$

the rules associated with the inductive types, for example:

$$\begin{array}{rcl} natrec(X, Y, 0) & \rightarrow & X \\ natrec(X, Y, s(n)) & \rightarrow & (Y\ n\ natrec(X, Y, n)) \end{array}$$

for the inductive type `nat`, and the rules used for defining the function symbols, for example:

$$\begin{array}{rcl} i(0_{\mathtt{nat}}) & \rightarrow & 0_{\mathtt{ord}} \\ i(s_{\mathtt{nat}}(x)) & \rightarrow & s_{\mathtt{ord}}(i(x)) \end{array}$$

for the injection function from `nat` to `ord`. We can immediately see that the recursor rules look very much like the rules defining the injection. We will show in Section 4 that the recursor rules for strictly positive inductive types follow the General Schema defined in Subsection 2.3 and, therefore, the recursor rules need not be singled out in our technical developments.

2.2 Higher-order rewriting

Before to define the General Schema precisely, we need to introduce the notion of higher-order rewriting that we use. Indeed, several notions of higher-order rewriting exist in the literature. Ours is the simplest possible: a term u

rewrites to a term u' by using a rule $l \to r$ if u *matches* the left-hand side l or, equivalently, if u is an *instance* of l by some substitution θ. Matching here is syntactic, that is, u is α-convertible to the instance of l. In contrast, the more sophisticated notions of higher-order rewriting defined by Klop (Combinatory Reduction Systems [30,31]), Nipkow (Higher-order Rewrite Systems [39,34]) and van Raamsdonk and van Oostrom (Higher-Order Rewriting Systems [49,50], generalizing both) are based on higher-order pattern-matching, that is, u must be $\beta\eta\alpha$-convertible to the instance of l.

Definition 6 (Rewrite rules and rewriting) *A* rewrite rule *is a pair $l \to r$ of terms such that:*

(1) l is headed by a function symbol,
(2) $FV(r) \subseteq FV(l)$,
(3) l and r have the same type.

Given a set R of rewrite rules, a term u R-rewrites to a term u' at position $p \in Pos(u)$ with the rule $l \to r \in R$, written $u \to_R^p u'$, if there exists a substitution θ such that $u|_p = l\theta$ and $u' = u[r\theta]_p$.

The defining rules *of a function symbol f are the rules whose left-hand side is headed by f.*

Condition (3) ensures that the reduction relation preserves types, that is, u and u' have the same type if $u \to_R u'$, a property called *subject reduction*.

We now give two more (classical) examples defining, for the first, the (formal) addition on Brouwer's ordinals and, for the second, some functions over lists. The first example is paradigmatic in its use of strictly positive types which are not basic. The second example uses a rule with an abstraction in the left-hand side. More complex examples of the second kind will be given in Section 4.

For the (formal) addition of Brouwer's ordinals,

$$\begin{aligned} x + 0 &\to x \\ x + s(y) &\to s(x + y) \\ x + lim(F) &\to lim(ln.(x + (F\ n))) \end{aligned}$$

note that the first two rules are just a first-order ones, hence a special case of higher-order rule. More important, note the need of an abstraction in the right-hand side of the last rule to bind the variable n needed for using the higher-order variable F taken from the left-hand side. This makes the termination proof of this set of rules a difficult task. In our case, the termination property will be readily obtained by showing that these rules follow our (improved) definition of the General Schema. The difficulty, of course, is simply delegated

to the strong normalization proof of the schema.

About Brouwer's ordinals [45], note that only a suitable choice of F's provides a semantically correct ordinal notation and that, for such a correct notation, the above formal definition provides semantically correct ordinal addition.

For the functions overs lists,

$$\begin{aligned} append(nil, l) &\rightarrow l \\ append(cons(x, l), l') &\rightarrow cons(x, append(l, l')) \\ append(append(l, l'), l'') &\rightarrow append(l, append(l', l'')) \end{aligned}$$

$$\begin{aligned} map(F, nil) &\rightarrow nil \\ map(F, cons(x, l)) &\rightarrow cons((F\ x), map(F, l)) \\ map(F, append(l, l')) &\rightarrow append(map(F, l), map(F, l')) \\ map(lx.x, l) &\rightarrow l \end{aligned}$$

note that the three first rules, which define the concatenation *append* of two lists, are again usual first-order rules. The four next rules define the function *map* which successively applies the function F to the elements of some list. Note that the third and sixth rule use a matching over a function symbol, namely *append*.

2.3 The General Schema

We now proceed to describe the schema that the user-defined higher-order rules should follow. In particular, all examples of higher-order rules given so far satisfy this schema. It is inspired from the last two authors former General Schema [27,28] although the formulation is quite different. The new schema is more powerful and answers a problem left open with the former one, that is, the ability of capturing definitions like the one previously given for the addition on ordinals. The main property of the schema is that it ensures the termination property of the relation $\rightarrow_R \cup \rightarrow_{\beta\eta}$, for any set R of rules following the General Schema. This will be the subject of Section 3.

In a function definition, in the case of a recursive call, we need a way to compare the arguments of the recursive calls in the right-hand side with the arguments of the left-hand side, and prove that they strictly decrease to ensure termination. What we expect to use as the comparison ordering is the subterm ordering or some extension of it. The one we are going to introduce is similar to Coquand's notion of "structurally smaller" [12] and will allow us to deal

with definitions like the addition on ordinals. The comparison between the recursive call arguments and the left-hand side arguments will then be done in a lexicographic or multiset manner, or a combination thereof, according to a *status* of the function symbol being defined. This status can be given by the user, or computed in non-deterministic linear time.

In the following, we assume given a family $\{x_i\}_{i\geq 1}$ of variables.

Definition 7 (Status ordering) *A* status *is a linear term* $stat = lex(u_1, \ldots, u_p)$ $(p \geq 1)$ *where each* u_i *is of the form* $mul(x_{k_1}, \ldots, x_{k_q})$ $(q \geq 1)$ *with* $x_{k_1}, \ldots, x_{k_q}$ *of the same type. The* arity *of stat is the greatest indice* i *such that* x_i *occurs in stat. The set* $Lex(stat)$ *of* lexicographic positions *in stat is the set of indices* i *such that there exists* $j \in \{1, \ldots, p\}$ *for which* $u_j = mul(x_i)$*, that is,* $q = 1$*.*

Given a status stat of arity n*, a strict ordering* $>$ *on a set* E *can be extended to an ordering* $>_{stat}$ *on sequences of elements of* E *of length greater or equal to* n *as follows:*

- $\vec{u} >_{stat} \vec{v}$ *iff* $stat\{\vec{x} \mapsto \vec{u}\} >_{stat}^{lex} stat\{\vec{x} \mapsto \vec{v}\}$
- $lex(\vec{u}) >_{stat}^{lex} lex(\vec{v})$ *iff* $\vec{u}(>_{stat}^{mul})_{lex}\vec{v}$
- $mul(\vec{u}) >_{stat}^{mul} mul(\vec{v})$ *iff* $\{\vec{u}\} >_{mul} \{\vec{v}\}$

where $>_{lex}$ *and* $>_{mul}$ *denote the lexicographic and multiset extension of* $>$ *respectively.*

For example, with $stat = lex(x_3, mul(x_2, x_4))$, $\vec{u} \rhd_{stat} \vec{v}$ iff $u_3 \rhd v_3$ or else $u_3 = v_3$ and $\{u_2, u_4\} \rhd_{mul} \{v_2, v_4\}$. Note that a status ordering *stat* boils down to the usual lexicographic ordering if $stat = lex((mul(x_1), \ldots, mul(x_n))$ or to the multiset ordering if $stat = lex(mul(x_1, \ldots, x_n))$. An important property of status orderings is that $>_{stat}$ is well-founded if $>$ is well-founded.

The notion of status will allow us to accept definitions like the ones below. For the Ackermann function Ack, we need to take the lexicographic status $stat_{Ack} = lex(mul(x_1), mul(x_2))$ and, for the binomial function $Bin(n, m) = C_{m+n}^{n}$, we need to take the multiset status $stat_{Bin} = lex(mul(x_1, x_2))$.

$$\begin{aligned} Ack(0, y) &\rightarrow s(y) \\ Ack(s(x), 0) &\rightarrow Ack(x, s(0)) \\ Ack(s(x), s(y)) &\rightarrow Ack(x, Ack(s(x), y)) \end{aligned}$$

$$
\begin{aligned}
Bin(0, m) &\rightarrow s(0) \\
Bin(s(n), 0) &\rightarrow s(0) \\
Bin(s(n), s(m)) &\rightarrow Bin(n, s(m)) + Bin(s(n), m)
\end{aligned}
$$

Apart from the notion of status, the other ingredients of our schema are new. We introduce them in turn.

Definition 8 (Symbol definitions) *We assume that each function symbol f of arity $n \geq 1$ comes along with a status $stat_f$ of arity p such that $1 \leq p \leq n$ and a set R_f of rewrite rules defining f. We denote by R the set of all rewrite rules and by $\rightarrow = \rightarrow_R \cup \rightarrow_{\beta\eta}$ the rewrite relation of the calculus.*

Assumption 2: We assume that the precedence $>_{\mathcal{F}}$ is well-founded and that $stat_f = stat_g$ whenever $f =_{\mathcal{F}} g$.

The main new idea in the definition of the General Schema is to construct a set of admissible right-hand sides, once a left-hand side is given. This set will be generated inductively from a starting set of terms extracted from the left-hand side, called the set of *accessible* subterms, by the use of *computability* preserving operations. Here, computability refers to Tait's computability predicate method for proving the termination of the simply-typed l-calculus [46], which was later extended by Girard to the polymorphic l-calculus [22,24].

To explain our construction, we need to recall the basics of Tait's method. The starting observation is that it is not possible to prove the termination of β-reduction directly by induction on the structure of terms because of the application case: in the untyped l-calculus, the term $(lx.xx\ lx.xx)$ rewrites to itself although $lx.xx$ is in normal form. Tait's idea was to strengthen the induction hypothesis by using instead a property, the *computability*, implying termination. The computability predicate can be defined by induction on the type of terms as follows: for an inductive type $\mathtt{s}$, take $[\![\mathtt{s}]\!] = SN^{\mathtt{s}}$, the set of strongly normalizable terms of type $\mathtt{s}$ (terms having no infinite sequence of rewrites issued from them). For a functional type $s \rightarrow t$, take $[\![s \rightarrow t]\!] = \{u \in \text{Ł}^{s \rightarrow t} \mid \forall v \in [\![s]\!],\ (u\ v) \in [\![t]\!]\}$. From this definition, it is easy to prove that every computable term is strongly normalizable ($[\![s]\!] \subseteq SN^s$) and that every term is computable ($\text{Ł}^s \subseteq [\![s]\!]$). Therefore, every term is strongly normalizable. The role of the General Schema when rewrite rules are added is to ensure that computability is preserved along the added rewritings. This is why we require that a right-hand side of rule is built up from subterms of the left-hand side, the *accessible* ones, by computability preserving operations: a set called the *computable closure* of the left-hand side.

Definition 9 (Accessible subterms) *Given a term v, the set $Acc(v)$ of* accessible *subterms of v is inductively defined as follows:*

(1) $v \in Acc(v)$,
(2) *if* $lx.u \in Acc(v)$ *then* $u \in Acc(v)$,
(3) *if* $C(\vec{u}) \in Acc(v)$ *then each* $u_i \in Acc(v)$,
(4) *if* $(u\ x) \in Acc(v)$ *and* $x \notin FV(u) \cup FV(v)$ *then* $u \in Acc(v)$,
(5) *if* u *is a subterm of* v *of basic type such that* $Fv(u) \subseteq FV(v)$ *then* $u \in Acc(v)$.

To see how this works, let us consider the examples of *append* and *map* given in Subsection 2.2. For the rule $append(nil, l) \to l$, l is accessible in the arguments of *append* by (1). For the rule $append(cons(x, l), l') \to cons(x, append(l, l'))$, l is accessible in $cons(x, l)$ by (3) and (1). The other rules are dealt with in the same way. Another example is given by the associativity rule of the addition on natural numbers: in the rule $(x + y) + z \to x + (y + z)$, the variables x and y are accessible by (5). This does not work for the addition on Brouwer's ordinals since `ord` is not a basic inductive type. The cases (2) and (4) will be useful in the more complex examples of Section 4.

We have already seen how to extract subterms from a left-hand side of rule. We are left with the construction of the computable closure from these subterms. Among the operations used for the computable closure, one constructs recursive calls with "smaller" arguments. We therefore need to define the intended ordering, which has to be richer than the usual subterm ordering as examplified by the last rule of the definition of the addition on Brouwer's ordinals:

$$x + lim(F) \ \to \ lim(ln.(x + (F\ n)))$$

We see that $(F\ n)$, the second argument of the recursive call, is not a strict subterm of $lim(F)$. Extending the General Schema so as to capture such definitions was among the open problems mentioned in [28]. On the other hand, in a set-theoretic interpretation of functions as input-output pairs, the pair $(n, (F\ n))$ would belong to F, and therefore, $(F\ n)$ would in this sense be smaller than F. This is what is done by Coquand with his notion of "structurally smaller" [12] which he assumes to be well-founded without a proof. Here, we make the same idea more concrete by relating it to the strict positivity condition of inductive types.

Definition 10 (Ordering on arguments) *Let* s *be a type and* u *and* v *be two terms of type* s.

- *If* s *is a strictly positive inductive type then* u *is* greater than v, $u > v$, *if there is* $p \in Pos(u)$ *such that* $p \neq \varepsilon$, $v = (u|_p\ \vec{v})$ *and, for all* $q < p$, $u|_q$ *is constructor-headed.*
- *Otherwise,* $u > v$ *if* v *is a strict subterm of* u *such that* $FV(v) \subseteq FV(u)$.

We are now ready to define the *computable closure* of a left-hand side.

Definition 11 (Computable closure) *Given a symbol $f \in \mathcal{F}_{s_1,\ldots,s_n,s}$, the* computable closure $\mathcal{CC}_f(\vec{l})$ *of some term $f(\vec{l})$ is inductively defined as the least set $\mathcal{CC}$ such that:*

(1) if x is a variable then $x \in \mathcal{CC}$,
(2) if $u \in Acc(\vec{l})$ then $u \in \mathcal{CC}$,
(3) if u and v are two terms in $\mathcal{CC}$ of respective types $t_1 \to t_2$ and t_1 then $(u\ v) \in \mathcal{CC}$,
(4) if $u \in \mathcal{CC}$ then $lx.u \in \mathcal{CC}$,
(5) if $g \in \mathcal{F}_{t_1,\ldots,t_p,t}$, $g <_{\mathcal{F}} f$ and $u_1, \ldots, u_p$ are p terms in $\mathcal{CC}$ of respective types $t_1, \ldots, t_p$ then $g(\vec{u}) \in \mathcal{CC}$,
(6) if $g \in \mathcal{F}_{t_1,\ldots,t_p,t}$, $g =_{\mathcal{F}} f$ and $u_1, \ldots, u_p$ are p terms in $\mathcal{CC}$ of respective types $t_1, \ldots, t_p$ then $g(\vec{u}) \in \mathcal{CC}$ whenever:
- *$\vec{l} >_{stat_f} \vec{u}$,*
- *if $l_i > (l_i|_p\ \vec{v})$ then each v_i belongs to $\mathcal{CC}$.*

Definition 12 (General Schema) *A rewrite rule $f(\vec{l}) \to r$ follows the* General Schema (GS) *if $r \in \mathcal{CC}_f(\vec{l})$ and, for every $x \in FV(r)$, $x \in Acc(\vec{l})$.*

As an example, let us prove that the definitions of *append* and *map* given in Subsection 2.2 indeed follow the General Schema. We already saw that the free variables occuring in the left-hand sides were all accessible hence, by (2), they belong to the computable closure (CC) of their respective left-hand side. For the rule $append(cons(x, l), l') \to cons(x, append(l, l'))$, $append(l, l')$ belongs to (CC) by (6) since l is a strict subterm of $cons(x, l)$. For the rule $map(F, cons(x, l)) \to cons((F\ x), map(F, l))$, $(F\ x)$ belongs to (CC) by (3), $map(F, l)$ by (6) and the whole right-hand side by (5). The other rules are dealt similarly.

In our previous definition of the General Schema, the computable closure was kind of implicit with, in particular, a poor accessibility relation and a case (7) in which the ordering used was always the strict subterm ordering.

The main differences with Coquand's notion of "structurally smaller" [12] or its extension by Giménez [20] are that:

(1) we use statuses for comparing the arguments of the recursive calls with the left-hand side arguments (which include lexicographic comparisons),
(2) we may compare a function-headed term or a l-headed term with one of its subterm while, in Coquand's definition, comparisons are restricted to constructor-headed terms.

A main advantage of both notions of accessibility and computable closure is

their formulation: it is immediate to add new cases in these definitions. This flexibility should of course be essential when extending the schema to richer calculi.

Given a user's specification following the General Schema, the question arises whether the following properties are satisfied: subject reduction, confluence, completeness of definitions and strong normalization. Subject reduction follows easily. Confluence reduces to local confluence once strong normalization is satisfied and can therefore be checked on the critical pairs. Completeness of definitions is necessary for the recursor definitions to make sense in our Curry-Howard interpretation of types. Checking it can be done by solving (higher-order) disequations. As recalled in [28], this can be done automatically for a reasonable fragment of the set of second order terms. In the next section, we address the remaining problem, strong normalization.

3 Strong normalization

In this section, we prove that the rewrite relation $\rightarrow = \rightarrow_R \cup \rightarrow_{\beta\eta}$ is terminating, i.e. there is no infinite sequence of rewrites, whenever all rules of R satisfy the General Schema. Due to the formulation of the schema, our proof here is much simpler than the one in [28], although the schema is more general. It is again based on Tait's computability predicate method. See [19] for a comprehensive survey of the method.

We first define the interpretation of types and prove important properties about it. In a second part, we prove a computability property for the function symbols: assuming that the rules satisfy the General Schema, a term headed by a function symbol is computable whenever its arguments are computable. Strong normalization follows then easily.

3.1 Interpretation of types

Definition 13 (Interpretation of types) *The interpretation $[\![s]\!]$ of a type $s \in \mathcal{T}$ is inductively defined as follows:*

- $[\![\mathtt{s} \in \mathcal{I}]\!]$ *is the set of terms $u \in SN^{\mathtt{s}}$ such that, for all term $C(\vec{u})$ such that $u \rightarrow^* C(\vec{u})$, each $u_i \in [\![s_i]\!]$,*
- $[\![s \rightarrow t]\!] = \{u \in \mathcal{L}^{s \rightarrow t} \mid \forall v \in [\![s]\!],\ (u\ v) \in [\![t]\!]\}$.

In the following, we will say that a term of type s is computable *if it belongs to $[\![s]\!]$ and that a substitution θ is* computable *if, for every variable $x \in dom(\theta) \cap \mathcal{X}^s$, $x\theta \in [\![s]\!]$.*

The reason why we need such a complex interpretation is because we need the property that the arguments of a computable constructor-headed term are computable. Meanwhile, we will see in Lemma 16.7 just below that, in case of a basic inductive type $\mathtt{s}$, the interpretation is merely $SN^{\mathtt{s}}$.

But, first, we show that our definition makes sense.

Lemma 14 *For every type $s \in \mathcal{T}$, $[\![s]\!]$ is uniquely defined.*

PROOF. It suffices to prove that it holds for every inductive type $\mathtt{s} \in \mathcal{I}$. For the sake of simplicity, we assume that $=_{\mathcal{I}}$ is the identity, that is, there is no mutually inductive types. At the end, we tell how to treat the general case which, apart from the notations, is no more difficult. Let $\mathcal{P}(SN^{\mathtt{s}})$ be the set of subsets of $SN^{\mathtt{s}}$. $\mathcal{P}(SN^{\mathtt{s}})$ is a complete lattice with respect to set inclusion $\subseteq$. We show that $[\![\mathtt{s}]\!]$ is uniquely defined as the least fixpoint of a monotone functional over this lattice. The proof is by induction on $>_{\mathcal{I}}$ which is assumed to be well-founded.

We define the following family of functions $F_{\mathtt{s}} : \mathcal{P}(SN^{\mathtt{s}}) \to \mathcal{P}(SN^{\mathtt{s}})$ indexed by inductive types:

$$F_{\mathtt{s}}(X) = X \ \cup \ \left\{ u \in SN^{\mathtt{s}} \;\middle|\; \text{if } u \to^* C(\vec{u}) \text{ then each } u_i \in R_{s_i}(X) \right\},$$

$$\text{where } R_t(X) = \begin{cases} [\![\mathtt{t}]\!] & \text{if } \ t = \mathtt{t} \in \mathcal{I} \text{ and } \mathtt{s} >_{\mathcal{I}} \mathtt{t} \\ X & \text{if } \ t = \mathtt{s} \\ R_{t_1}(X) \to R_{t_2}(X) & \text{if } \ t = t_1 \to t_2 \end{cases}$$

Since inductive types are assumed to be (strictly) positive, $F_{\mathtt{s}}$ is monotone. Hence, from Tarski's theorem, it has a least fixed point, $[\![\mathtt{s}]\!]$.

In case of mutually inductive types, the function $F_{\mathtt{s}}$ operates on a product of subsets of $SN^{\mathtt{s}_1} \times \ldots \times SN^{\mathtt{s}_n}$ if $\mathtt{s}_1, \ldots, \mathtt{s}_n$ are all the inductive types equivalent to $\mathtt{s}$, which is again a lattice. Apart from the notations, the argument is therefore the same.

We showed that each $[\![\mathtt{s}]\!]$ is the least fixpoint of the monotone functional $F_{\mathtt{s}}$. This least fixpoint can be reached by transfinite iteration. Let $F_{\mathtt{s}}^{\mathfrak{a}}$ be the $\mathfrak{a}$-th iterate of $F_{\mathtt{s}}$ over the empty set. Note that we need to go further than ω as it is shown by the following example. Consider the function $f : \mathtt{nat} \to \mathtt{ord}$ defined by the following rules:

$$\begin{aligned} f(0_{\mathtt{nat}}) &\to 0_{\mathtt{ord}} \\ f(s_{\mathtt{nat}}(n)) &\to lim(lx.f(n)) \end{aligned}$$

For all n, $f(n) \in F^{n+1}_{\mathtt{ord}} \setminus F^{n}_{\mathtt{ord}}$. Thus, $lim(lx.f(x)) \in F^{\omega+1}_{\mathtt{ord}} \setminus F^{\omega}_{\mathtt{ord}}$.

This provides us a well-founded ordering on the computable terms of type $\mathtt{s}$:

Definition 15 (Ordering on the arguments of a function symbol) *The* order *of a term $t \in [\![\mathtt{s}]\!]$ is the smallest ordinal $\mathfrak{a}$ such that $t \in F^{\mathfrak{a}}_{\mathtt{s}}$. We say that $t \in [\![s]\!]$ is* greater than *$u \in [\![s]\!]$, $t \succ u$, if:*

- *$s \in \mathcal{I}$ and the order of t is greater than the order of u,*
- *$s = s_1 \to s_2$ and $t \to \cup \rhd u$.*

This is this ordering which allows us to treat the definitions on strictly positive types. This idea is already used by Mendler for proving the strong normalization of System F with recursors for positive inductive types [36] and by Werner for proving the strong normalization of the Calculus of Inductive Constructions with recursors for strictly positive types [51]. We apply this technique to a larger class of higher-order rewrite rules.

Let us see the example of the addition on Brouwer's ordinals. If $lim(f)$ is computable then, by definition of $[\![\mathtt{ord}]\!]$, f is computable. This means that, for any $n \in [\![\mathtt{nat}]\!]$, $(f\ n)$ is computable. Therefore, $lim(f) \succ (f\ n)$.

Lemma 16 (Computability properties) *A term is* neutral *if it is not an abstraction nor constructor-headed.*

(1) Every computable term is strongly normalizable.
(2) Every strongly normalizable term of the form $(x\ \vec{u})$ is computable.
(3) A neutral term is computable if all its immediate reducts are computable.
(4) $(lx.u\ v)$ is computable if v is strongly normalizable and $u\{x \mapsto v\}$ is computable.
(5) A constructor-headed term $C(\vec{u})$ is computable if the terms in $\vec{u}$ and all the immediate reducts of $C(\vec{u})$ are computable.
(6) Computability is preserved by reduction.
(7) If $\mathtt{s} \in \mathcal{I}$ is a basic inductive type then $[\![\mathtt{s}]\!] = SN^{\mathtt{s}}$.

PROOF.

(1) and (2) are proved together by induction on the type s of the term.
$s = \mathtt{s} \in \mathcal{I}$**:**
(1) $[\![\mathtt{s}]\!] \subseteq SN^{\mathtt{s}}$ by definition.

(2) Every strongly normalizable term $(x\ \vec{u})$ of type $\mathtt{s}$ is computable since it cannot reduce to a constructor-headed term.

$s = s_1 \to s_2$**:**

(1) Let u be a computable term of type s and x be a variable of type s_1. By induction hypothesis, $x \in [\![s_1]\!]$ hence, by definition of the interpretation for s, $(u\ x) \in [\![s_2]\!]$. By induction hypothesis again, $(u\ x) \in SN^{s_2}$. Therefore, $u \in SN^{s_1 \to s_2}$.

(2) Let $(x\ \vec{u})$ be a strongly normalizable term of type s and let $v \in [\![s_1]\!]$. By induction hypothesis, $v \in SN^{s_1}$ and $(x\ \vec{u}\ v) \in [\![s_2]\!]$. Therefore, $(x\ \vec{u}) \in [\![s_1]\!]$.

(3) is proved again by induction on the type s of the term.

$s = \mathtt{s} \in \mathcal{I}$**:**

Let u be a neutral term of type $\mathtt{s}$ whose immediate reducts belong to $[\![\mathtt{s}]\!]$. By (1), its immediate reducts are strongly normalizable, hence $u \in SN^{\mathtt{s}}$. Suppose now that u reduces to a constructor-headed term $C(\vec{v})$. Since u is neutral, it cannot be itself constructor-headed. Hence, $C(\vec{v})$ is a reduct of some immediate reducts u' of u. By definition of $\mathtt{s}$ and since $u' \in [\![\mathtt{s}]\!]$ by assumption, the terms in $\vec{v}$ are computable. Therefore $u \in [\![\mathtt{s}]\!]$.

$s = s_1 \to s_2$**:**

Let u be a neutral term of type s whose immediate reducts are computable and let $v \in [\![s_1]\!]$. By (1), $v \in SN^{s_1}$. Therefore, $\to$ is well-founded on the set of reducts of v.

Then, we prove that the immediate reducts of $(u\ v)$ belong to $[\![s_2]\!]$, by induction on v w.r.t. $\to$. As u is neutral, an immediate reduct of $(u\ v)$ is either of the form $(u'\ v)$ where u' is a reduct of u, or else of the form $(u\ v')$ where v' is a reduct of v. In the first case, since u' is computable by assumption, $(u'\ v) \in [\![s_2]\!]$. In the second case, we conclude by induction hypothesis on v'.

As a consequence, since $(u\ v)$ is neutral, by induction hypothesis, $(u\ v) \in [\![s_2]\!]$. Therefore, u is computable.

(4) Since $(lx.u\ v)$ is neutral, by (3), it suffices to prove that each one of its reducts is computable. The reduct $u\{x \mapsto v\}$ is computable by assumption. Otherwise, we reason by induction on the set of the reducts of u and v (which are both strongly normalizable) with $\to$ as well-founded ordering.

(5) Let $C(\vec{u})$ be a constructor-headed term such that the terms in $\vec{u}$ and all its immediate reducts are computable. Then, it is strongly normalizable since, by (1), all its immediate reducts are strongly normalizable. Now, let $D(\vec{v})$ be a constructor-headed term such that $C(\vec{u}) \to^* D(\vec{v})$. If $D(\vec{v}) = C(\vec{u})$ then the terms in $\vec{v} = \vec{u}$ are computable by assumption. Otherwise, there is an immediate reduct v of $C(\vec{u})$ such that $v \to^* D(\vec{v})$. Since, by assumption, v is computable, the terms in $\vec{v}$ are computable. Hence, $C(\vec{u})$ is computable.

(6) is proved again by induction on the type s of the term.

$s = \mathtt{s} \in \mathcal{I}$**:**

Let $u \in [\![\mathtt{s}]\!]$ and u' be a reduct of u. By (1), $u \in SN^{\mathtt{s}}$, hence $u' \in SN^{\mathtt{s}}$. Besides, if u' reduces to a constructor-headed term $C(\vec{v})$ then u reduces to

$C(\vec{v})$ as well. Therefore, by definition of $[\![\mathtt{s}]\!]$, the terms in $\vec{v}$ are computable and $u' \in [\![\mathtt{s}]\!]$.

$s = s_1 \to s_2$**:**
Let u be a computable term of type s, u' be a reduct of u and $v \in [\![s_1]\!]$. $(u'\ v)$ is a reduct of $(u\ v)$ which, by definition of $[\![s]\!]$, belongs to $[\![s_2]\!]$. Hence, by induction hypothesis, $(u'\ v) \in [\![s_2]\!]$ and u' is computable.

(7) By (1), $[\![\mathtt{s}]\!] \subseteq SN^{\mathtt{s}}$. We prove that $SN^{\mathtt{s}} \subseteq [\![\mathtt{s}]\!]$, by induction on $SN^{\mathtt{s}}$ with $\to \cup \rhd$ as well-founded ordering. Let $u \in SN^{\mathtt{s}}$ and suppose that $u \to^* C(\vec{v})$ where $C \in \mathcal{C}(\mathtt{s})$. Since $\mathtt{s}$ is basic, $\tau(C) = \mathtt{s}_1 \to \ldots \to \mathtt{s}_n \to \mathtt{s}$ where each $\mathtt{s}_i$ is also a basic inductive type. Each v_i is strongly normalizable hence, by induction hypothesis, each v_i is computable. Therefore, u is computable.

3.2 Computability of function symbols

We start this paragraph by proving that accessibility is compatible with computability, that is, any term accessible in a computable term is computable. Then, we prove the same property for the computable closure.

Lemma 17 (Compatibility of accessibility with computability) *Let v be a term and θ a computable substitution such that $dom(\theta) \subseteq FV(v)$ and $v\theta$ is computable. If u is accessible in v and θ' is a computable substitution such that $dom(\theta') \cap FV(v) = \emptyset$ then $u\theta\theta'$ is computable.*

PROOF. By induction on $u \in Acc(v)$.

(1) The case $u = v$ is immediate since $u\theta\theta' = v\theta\theta' = v\theta$.

(2) $lx.u \in Acc(v)$. $\theta' = \theta'' \uplus \{x \mapsto x\theta'\}$ with $x \notin dom(\theta'')$. $u\theta\theta' = u\theta\theta''\{x \mapsto x\theta'\}$ is a reduct of $(lx.u\theta\theta''\ x\theta')$. $dom(\theta'') \cap FV(v) = \emptyset$ hence, by induction hypothesis, $lx.u\theta\theta''$ is computable. Therefore, $u\theta\theta'$ is computable since, by assumption on θ', $x\theta'$ is computable.

(3) $u = u_i$ and $C(\vec{u}) \in Acc(v)$. By induction hypothesis, $C(\vec{u}\theta\theta')$ is computable. Therefore, by definition of computability for inductive types, $u\theta\theta'$ is computable.

(4) $(u\ x) \in Acc(v)$ and $x \notin FV(u) \cup FV(v)$. u must be of type $s \to t$ and $x \notin dom(\theta')$. Then, let w be a computable term of type s and $\theta'' = \theta' \uplus \{x \mapsto w\}$. $dom(\theta'') \cap FV(v) = \emptyset$ hence, by induction hypothesis, $(u\ x)\theta\theta'' = (u\theta\theta'\ w)$ is computable. Therefore $u\theta\theta'$ is computable.

(5) u is a subterm of v of basic type such that $FV(u) \subseteq FV(v)$. Since $FV(u) \subseteq FV(v)$, $u\theta\theta' = u\theta$ is a subterm of $v\theta$. Since $v\theta$ is computable, hence strongly normalizable, its subterm $u\theta$ is also strongly normalizable, hence computable, since it is of basic type.

Lemma 18 (Computability of function symbols) *Assume that the rules of R satisfy the General Schema. For every function symbol f, if $f(\vec{u})$ is a term whose arguments are computable, then $f(\vec{u})$ is computable.*

PROOF. The proof uses three levels of induction: on the function symbols ordered by $>_{\mathcal{F}}$ (H1), on the arguments of f (H2) and on the right-hand side structure of the rules defining f (H3).

After Lemma 16.3 and 16.5 (the terms in $\vec{u}$ are computable by assumption), $f(\vec{u})$ is computable if all its immediate reducts w are computable. We prove that by induction on $(\vec{u}, \vec{u})$ with $(\succ_{stat_f}, \rightarrow_{lex})_{lex}$ as well-founded ordering (H2).

If the reduction does not take place at the root, then $w = f(\vec{u'})$ with $\vec{u} \rightarrow_{lex} \vec{u'}$. Since computability predicates are stable by reduction, the terms in $\vec{u'}$ are computable. Now, it is not difficult to see that $\succ$ is compatible with $\rightarrow$, that is, $u \succeq u'$ whenever $u \rightarrow u'$. Hence, by induction hypothesis (H2), w is computable.

If the reduction takes place at the root, then there are a rule $f(\vec{l}) \rightarrow r$ and a substitution θ such that $dom(\theta) = FV(\vec{l})$, $\vec{u} = \vec{l}\theta$ and $w = r\theta$. By definition of the General Schema, every variable x free in r is accessible in $\vec{l}$. Hence, by Lemma 17 (take the identity for θ'), for all $x \in FV(r)$, $x\theta$ is computable since, by hypothesis, the terms in $\vec{l}\theta = \vec{u}$ are computable. Therefore the substitution $\theta|_{FV(r)}$ is computable.

We now show by induction on $r \in \mathcal{CC}_f(\vec{l})$ that, for any computable substitution θ' such that $dom(\theta') \cap FV(r) = \emptyset$, $r\theta\theta'$ is computable (H3).

(1) r is a variable x. If $x \in dom(\theta\theta')$ then $r\theta\theta' = x\theta\theta'$ is computable since $\theta\theta'$ is computable. If $x \notin dom(\theta\theta')$ then $r\theta\theta' = x$ is computable since any variable is computable.
(2) $r \in Acc(\vec{l})$. By Lemma 17.
(3) $r = (v\ w)$ with v and w in $\mathcal{CC}_f(\vec{l})$. By induction hypothesis (H3), $v\theta\theta'$ and $w\theta\theta'$ are computable. Therefore, by definition of computability predicates, $r\theta\theta'$ is computable.
(4) $r = lx.v$ with $v \in \mathcal{CC}_f(\vec{l})$. Let $s \rightarrow t$ be the type of r and w be a computable term of type s. By induction hypothesis (H3), $v\theta\theta'\{x \mapsto w\}$ is computable. Hence, by Lemma 16.4, $r\theta\theta'$ is computable.
(5) $r = g(\vec{v})$ with $g <_{\mathcal{F}} f$ and each $v_i \in \mathcal{CC}_f(\vec{l})$. By induction hypothesis (H3), each $v_i\theta\theta'$ is computable. Hence, by induction hypothesis (H1), $r\theta\theta'$ is computable.
(6) $r = g(\vec{v})$ with $g =_{\mathcal{F}} f$, each $v_i \in \mathcal{CC}_f(\vec{l})$ and $\vec{l} >_{stat_f} \vec{v}$. By induction hypothesis (H3), each $v_i\theta\theta'$ is computable. We show that $\vec{l}\theta\theta' \succ_{stat_f} \vec{v}\theta\theta'$.
   - Assume that $l_i > v_j$ and l_i is of type a strictly positive inductive type $\mathtt{s}$.

By definition of $>$, there is $p \in Pos(l_i)$ such that $p \neq \varepsilon$, $v_j = (l_i|_p\ \vec{v})$ and, for all $q < p$, $l_i|_q$ is constructor-headed. By assumption, each v_i belongs to the computable closure. So, by induction hypothesis (H3), $v_i\theta\theta'$ is computable. Now, $l_i|_p$ has a type of the form $\vec{s} \to \mathtt{s}$. Let $\mathtt{s}_q$ be the type of $l_i|_q$. Since $>_{\mathcal{I}}$ is well-founded, all the $\mathtt{s}_q$'s are equivalent to $\mathtt{s}$. Thus, if $p = i_1 \dots i_{k+1}$ then $l_i\theta\theta' \succ l_i|_{i_1}\theta\theta' \succ \dots l_i|_{i_1 \dots i_k}\theta\theta' \succ (l_i|_p\theta\theta'\ \vec{v}\theta\theta')$.

- v_j is a strict subterm of l_i such that $FV(v_j) \subseteq FV(l_i)$. Hence, $v_j\theta\theta'$ is a strict subterm of $l_j\theta\theta'$ and $v_j\theta\theta' \succ l_j\theta\theta'$.
  Therefore, by induction hypothesis (H2), $r\theta\theta'$ is computable.

We are now able to prove the main lemma for strong normalization, i.e. every term is computable. The strong normalization itself will follow as a simple corollary.

Lemma 19 (Main lemma) *Assume that all the rules of R follow the General Schema. Then, for every term u and computable substitution θ, $u\theta$ is computable.*

PROOF. We proceed by induction on the structure of u.

(1) u is a variable x. If $x \in dom(\theta)$ then $u\theta = x\theta$ is computable since θ is computable. If $x \notin dom(\theta)$ then $u\theta = x$ is computable since any variable is computable.
(2) $u = f(\vec{u})$. By induction hypothesis, each $v_i\theta$ is computable. Therefore, by Lemma 18, $u\theta$ is computable.
(3) $u = lx.v$. Let $s \to t$ be the type of u, w be a computable term of type s and $\theta' = \theta \uplus \{x \mapsto w\}$. By induction hypothesis, $v\theta'$ is computable. Therefore, by Lemma 16.4, $(u\theta\ w)$ is computable and $u\theta$ also.
(4) $u = (v\ w)$. By induction hypothesis, $v\theta$ and $w\theta$ are computable. Therefore, by definition of computability, $u\theta$ is computable.

Theorem 20 (Strong normalization) *Under the assumptions 1 and 2, the combination of*

(1) the simply-typed l-calculus with $\beta\eta$-reductions and
(2) higher-order rewrite rules following the General Schema

is strongly normalizing.

PROOF. Since a computability predicate of type s contains all variables of type s, the identity substitution is computable. Hence, by Lemma 19, every term is computable. And since computable terms are strongly normalizable, every term is strongly normalizable.

4 Examples and Extensions

In this section, we present several applications and current limitations of the General Schema termination proof method.

4.1 Recursors for strictly positive types

We already saw that the addition on Brouwer's ordinals follows the General Schema. This is also true of the recursor on Brouwer's ordinals [45], as the user can easily check it:

$$\begin{aligned} ordrec_t(X,Y,Z,0) &\rightarrow X \\ ordrec_t(X,Y,Z,s(n)) &\rightarrow (Y\ n\ ordrec_t(X,Y,Z,n)) \\ ordrec_t(X,Y,Z,lim(F)) &\rightarrow (Z\ F\ ln.ordrec_t(X,Y,Z,(F\ n))) \end{aligned}$$

where $ordrec_t$ is of type $t \rightarrow (\texttt{ord} \rightarrow t \rightarrow t) \rightarrow ((\texttt{nat} \rightarrow \texttt{ord}) \rightarrow (\texttt{nat} \rightarrow t) \rightarrow t) \rightarrow \texttt{ord} \rightarrow t$.

This is true as well of the recursors on mutually inductive types, such as the type for trees:

$$\begin{aligned} treerec_t(X,Y,Z,node(l)) &\rightarrow (X\ l\ listtreerec_t(X,Y,Z,l)) \\ listtreerec_t(X,Y,Z,nil) &\rightarrow Y \\ listtreerec_t(X,Y,Z,cons(x,l)) &\rightarrow \\ &\quad (Z\ x\ l\ treerec_t(X,Y,Z,x)\ listtreerec_t(X,Y,Z,l)) \end{aligned}$$

The same property is actually true of arbitrary strictly positive inductive types. The general case is no more difficult apart for the more complex notations.

The uniqueness rules for recursors of basic inductive types were studied in [41] and extended to the strictly positive case in [26]. In both cases, the termination proof did not use the General Schema since the uniqueness rules do not seem to fit the General Schema. It is open whether one could modify the schema to cover this kind of rules.

4.2 Curried function symbols

We have assumed that all function symbols come along with all their arguments. This is due to the fact that η together with rewrite rules over curried symbols lead to non-confluence. Take for example $id : \texttt{nat} \to \texttt{nat}$ defined by $(id\ x) \to x$. Then, $lx.x \leftarrow lx.(id\ x) \to_\eta id$.

Using curried symbols, however, is possible to the price of duplicating the vocabulary as follows: for each function symbol f of arity $n > 0$, we add a new function symbol f^c of the same type as f but of arity 0, defined by the rule

$$f^c \ \to \ lx_1 \ldots lx_n.f(x_1, \ldots, x_n)$$

which satisfies the General Schema. Here is an example of definition of the sum of a list of natural numbers using the *foldl* function:

$$\begin{aligned} \mathit{foldl}(F, x, \mathit{nil}) &\to x \\ \mathit{foldl}(F, x, \mathit{cons}(y, l)) &\to \mathit{foldl}(F, (F\ x\ y), l) \\ +^c &\to lxy.x + y \\ \mathit{sum}(l) &\to \mathit{foldl}(+^c, 0, l) \end{aligned}$$

4.3 First-order rewriting

In [28], the last two authors proved that it was possible to combine higher-order rewrite rules following the General Schema with a first-order rewrite system whose rules decrease in some rewrite ordering and are non-duplicating (ie. no free variable occurs more often in the right-hand side than in the left-hand side), a condition needed to avoid Toyama's counter-example to the modularity of termination [47]. It is of course possible to do the same here, using Lemma 24 of [28], an analog of Lemma 18 for first-order functions symbols.

Below, we give an example which cannot be proved to terminate by our method: let $-$ and $/$ be the subtraction and division over natural numbers. Note that $-$ follows the General Schema while $/$ does not and that the last rule is duplicating the variable y:

$$\begin{aligned} 0 - y &\to 0 \\ s(x) - 0 &\to s(x) \\ s(x) - s(y) &\to x - y \end{aligned} \qquad \begin{aligned} x\ /\ 0 &\to x \\ 0\ /\ s(y) &\to 0 \\ s(x)\ /\ s(y) &\to s((x - y)\ /\ s(y)) \end{aligned}$$

In [21], Giménez proposes a terminating schema using a notion of subtyping which allows to prove the strong normalization property of this example.

However, we do not think this is a real issue. Non-termination does not necessarily imply logical inconsistence, i.e. *False* is provable. In the case of Toyama's counterexample to the modularity property of termination, the union of the two original confluent and terminating rewrite systems is not terminating, but every term has a computable normal form. We believe, hence conjecture, that this property is enough here to ensure that *False* cannot be derived in the combined calculus.

4.4 Conditional rewriting

A conditional rule is a triple written $(l \to r \text{ if } C)$ where C is a *condition* of the form $u_1 = v_1 \wedge \ldots \wedge u_n = v_n$ with $FV(C) \subseteq FV(l)$, meaning that $l \to r$ may be applied only if the terms of each pair (u_i, v_i) have a common reduct. The conditional rule:

$$l \to r \quad \text{if} \quad u_1 = v_1 \wedge \ldots \wedge u_n = v_n$$

can be encoded with the two non-conditional rules:

$$\begin{aligned} l &\to eq_n(u_1, v_1, \ldots, u_n, v_n, r) \\ eq_n(x_1, x_1, \ldots, x_n, x_n, z) &\to z \end{aligned}$$

The second rule satisfies the General Schema quite trivially. We therefore say that a conditional rule follows the General Schema if $l \to r$ follows the General Schema and $u_1, v_1, \ldots, u_n, v_n$ are all in the computable closure of l. Hence, after Theorem 20, if all the conditional rules satisfy the General Schema, then $\to \cup \to_{\beta\eta}$ is strongly normalizing.

A well-known example is given by an insertion function on lists.

$$\begin{aligned} insert(x, nil) &\to cons(x, nil) \\ insert(x, cons(y, l)) &\to cons(x, cons(y, l)) \quad \text{if} \quad inf(x, y) = true \\ insert(x, cons(y, l)) &\to cons(y, insert(x, l)) \quad \text{if} \quad inf(x, y) = false \\ inf(0, x) &\to true \\ inf(s(x), 0) &\to false \\ inf(s(x), s(y)) &\to inf(x, y) \end{aligned}$$

We are going to see that our method can easily cope with basic inductive types whose constructors satisfy some (first-order) equations, provided that these equations form a weakly-normalizing term rewriting system, that is, such that every term has a unique normal form. In this case, the initial algebra of the inductive type is equivalent to its normal form algebra and the latter can be represented by the accepting states of a finite tree automaton of some form [7,11]. The important property of this automaton is that the set of terms recognized at every accepting state is recursive and the predicate of this state is actually easy to define. We show the construction for the simple example of integers. The general case of an arbitrary basic inductive type is no different.

The inductive type `int` is specified with the constructors 0, s and p for zero, successor and predecessor respectively, and the two equations: $s(p(x)) = x$ and $p(s(x)) = x$, which are easily turned into a first-order convergent term rewriting system $\{s(p(x)) \to x,\ p(s(x)) \to x\}$ whose normal forms are recognized by the automaton given at Figure 1. This automaton can be easily constructed by solving disequations over terms (see [11,38]).

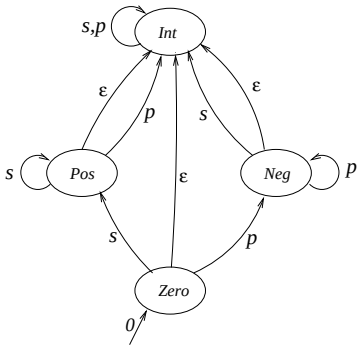

Fig. 1. Automaton

Then, the recursor on integers may be defined by the following set of constraint rules:

$$\begin{aligned} intrec_t(X, Y, Z, 0) &\to X \\ intrec_t(X, Y, Z, s(x)) &\to (Y\ x\ intrec(X, Y, Z, x)) \ \text{ if } \ s(x) \in Pos \\ intrec_t(X, Y, Z, p(x)) &\to (Z\ x\ intrec(X, Y, Z, x)) \ \text{ if } \ p(x) \in Neg \end{aligned}$$

As usual, it is then possible to define other functions such as the addition by the use of the recursor:

$$x + y \ \to \ intrec_{\texttt{int}}(x, lxy.s(y), lxy.p(y), y)$$

which is equivalent to the following pattern-matching definition:

$$
\begin{aligned}
x + 0 &\rightarrow x \\
x + s(y) &\rightarrow s(x + y) \text{ if } s(y) \in Pos \\
x + p(y) &\rightarrow p(x + y) \text{ if } p(y) \in Neg
\end{aligned}
$$

but to which we may add, for example, the rule for associativity:

$$
(x + y) + z \rightarrow x + (y + z)
$$

or, by a completely different definition which does not make use of the automaton but makes use of the signature present in the user's specification only:

$$
\begin{aligned}
x + 0 &\rightarrow x \\
x + s(y) &\rightarrow s(x + y) \\
x + p(y) &\rightarrow p(x + y)
\end{aligned}
\qquad\qquad
\begin{aligned}
s(p(x)) &\rightarrow x \\
p(s(x)) &\rightarrow x
\end{aligned}
$$

It is of course a matter of debate whether the normal form computations should be made available to the users, like the recursors, or should not. We have no definite argument in favor of either alternative.

We have assumed that the specification of constructors was a weakly normalizing (in practice, a confluent and terminating set) of rewrite rules. The method applies as well when some constructor is commutative or, associative and commutative (with some additional technical restriction). See [7] for more explanations and additional references. Whether it can be generalized to non-basic inductive types is however open.

4.6 Matching modulo βη

In this section, we address the case of higher-order rewrite rules *à la* Nipkow [39], based on higher-order pattern-matching with patterns *à la* Miller [37]. We give here several examples taken from [39], [48] or [42], and recall why plain pattern-matching does not really make sense for them. On the other hand, we will see that all these examples follow the General Schema: we explain the first example in detail and the user is invited to check the others against our definitions.

We start with the example of differentiation of functions over the inductive type `R`:

$$
\begin{array}{lll}
x \times 1 \;\to\; x & x \times 0 \;\to\; 0 & x + 0 \;\to\; x \\
1 \times x \;\to\; x & 0 \times x \;\to\; 0 & 0 + x \;\to\; x
\end{array}
\qquad 0 \;/\; x \;\to\; 0
$$

$$
\begin{array}{rcl}
D(lx.y) & \to & lx.0 \\
D(lx.x) & \to & lx.1 \\
D(lx.sin(F\ x)) & \to & lx.cos(F\ x) \times (D(F)\ x) \\
D(lx.cos(F\ x)) & \to & lx. - sin(F\ x) \times (D(F)\ x) \\
D(lx.(F\ x) + (G\ x)) & \to & lx.(D(F)\ x) + (D(G)\ x) \\
D(lx.(F\ x) \times (G\ x)) & \to & lx.(D(F)\ x) \times (G\ x) + (F\ x) \times (D(G)\ x) \\
D(lx.ln(F\ x)) & \to & lx.(D(F)\ x)\ /\ (F\ x)
\end{array}
$$

Note first that we cannot have composition explicitly as a constructor of the inductive type `R`, since the positivity condition would be violated. We could define it with the rule $F \circ G \to lx.(F\ (G\ x))$, but then, in $D(F \circ G)$, F and G are not accessible since they are not of basic type and, in $D(lx.(F\ (G\ x)))$, F is not accessible since it is not applied to distinct bound variables, a condition also required for patterns in Nipkow's framework. This explains why composition is encoded in each rule by using the application operator of the l-calculus.

The rules defining $\times$, $+$ and $/$ are usual first-order rules. We could restrict the use of the last one to the case where x is different from 0. Of course, this is not possible with a faithful axiomatization of reals, since equality to 0 is not decidable for the reals. As for the other rules, $D(lx.y) \to lx.0$ states that the differential of a constant function (equal to y) is the null function. The definition of substitution ensures here that x cannot occur freely in an instance of y, hence y is a constant with respect to x (although it may depend on other variables free in the rewritten term). The rule $D(lx.x) \to lx.1$ states that the differential of the identity is the constant function equal to 1. The next rule, $D(lx.sin(F\ x)) \to lx.cos(F\ x) \times (D(F)\ x)$, defines the differential of a function obtained by composing sin with some other function F. The other rules speak for themselves.

Assume now that we use first-order pattern-matching for these rules. Then, we would not be able to differentiate the function $lx.sin(x)$ by computing $D(lx.sin(x))$, because no rule would match. Of course, we could give new rules for this case, but this would be an endless game. The use of higher-order matching, on the other hand, chooses the appropriate value for the higher-order free variables so as to cover all cases.

The local confluence of these rules can be checked on higher-order critical

pairs, as shown by Nipkow [39,34]. The computation of these critical pairs can be done in linear time [43], thanks to the hypothesis that the left-hand sides are patterns.

We now show that this example follows the General Schema, by showing first that the free variables of the right-hand sides are accessible in their respective left-hand side. For the rule $D(lx.y) \to lx.0$, y is accessible in $lx.y$ by cases (1) and (2). For the rule $D(lx.sin(F\ x)) \to lx.cos(F\ x) \times (D(F)\ x)$, F is accessible in $lx.sin(F\ x)$ by (1), (2), (3) and (4). Now, it is not difficult to check that the right-hand sides belong to the computable closure of their respective left-hand side.

Prehofer and van de Pol prove the termination of this system (with higher-order pattern-matching) by defining a higher-order interpretation proved to be strictly monotonic on the positive natural numbers [42], a method developed by van de Pol [48] that generalizes to the higher-order case the interpretation method of first-order term rewriting systems. One can easily imagine that it is not easy at all to find higher-order interpretations. Here, D needs to be interpreted by a functional which takes as arguments a function f on positive natural numbers and a positive natural number n, for example the function $(f, n) \mapsto 1 + n \times f(n)^2$. Furthermore, the interpretation method is not modular, the adequate interpretation of each single function symbol depending on the whole set of rules. This makes it difficult to use by non-experts.

The next example is taken from process algebra [44]:

$$\begin{aligned}
p + p &\to p \\
(p + q)\,;r &\to (p\,;q) + (q\,;r) \\
(p\,;q)\,;r &\to p\,;(q\,;r) \\
p + \delta &\to p \\
\delta\,;p &\to \delta \\
\Sigma(ld.p) &\to p \\
\Sigma(X) + (X\ d) &\to \Sigma(X) \\
\Sigma(ld.(X\ d) + (Y\ d)) &\to \Sigma(X) + \Sigma(Y) \\
\Sigma(X)\,;p &\to \Sigma(ld.(X\ d)\,;p)
\end{aligned}$$

Note that the left-hand side of rule $\Sigma(X) + (X\ d)$ is not a pattern *à la* Miller. As a consequence, Nipkow's results for proving local confluence do not apply. Termination of these rules is also proved in [48]. To see that this example follows the General Schema, it suffices to take the precedence defined by $;> \delta, \Sigma$ and $\Sigma > +$. The rule $\Sigma(X) + (X\ d) \to \Sigma(X)$, which is a simple projection,

is dealt with by case (2).

The last example, the computation of the negative normal form of a formula, is taken from logic (we give only a sample of the rules):

$$\begin{aligned} \neg(\neg(p)) &\rightarrow p \\ \neg(p \wedge q) &\rightarrow \neg(p) \vee \neg(q) \\ \neg(\forall(P)) &\rightarrow \exists(lx.\neg(P\ x)) \end{aligned}$$

Of course, the fact that all the above examples follow the General Schema does not imply that Nipkow's rewriting terminates. However, we conjecture that it does and that it is due to the use of patterns in the left-hand sides. To prove our conjecture, we essentially need to show that higher-order pattern-matching preserves computability. This has been recently proved by the first author in [5], where the framework described here is extended into a typed version of Klop's higher-order rewriting framework [31], and where Nipkow's higher-order Critical Pair Lemma is shown to apply to this extended framework also.

4.7 Rewriting modulo additional theories

It is of general practice to rewrite modulo properties of constructors (implying that the underlying inductive type is a quotient) or defined symbols. Usual properties, as in presentations of arithmetic, are commutativity or, commutativity and associativity. In our encoding of predicate calculus, there is a less common kind of commutativity of bound variables, expressed by the equation:

$$\forall(lx.\forall(ly.(P\ x\ y))) = \forall(ly.\forall(lx.(P\ x\ y)))$$

We now give (a sample of) the rules for the computation of the prenex normal form of a formula:

$$\begin{aligned} \forall(P) \wedge q &\rightarrow \forall(lx.(P\ x) \wedge q) \\ p \wedge \forall(Q) &\rightarrow \forall(lx.p \wedge (Q\ x)) \end{aligned}$$

The above set of rules is confluent modulo the previous equation (but would not be confluent directly). Note that matching modulo the equation is not necessary here because of the form of the left-hand sides of rules.

We end this list with "miniscoping", an operation inverse of the prenex normal form:

$$\begin{aligned}
\forall(lx.p) &\rightarrow p \\
\forall(lx.(P\ x) \wedge (Q\ x)) &\rightarrow \forall(P) \wedge \forall(Q) \\
\forall(lx.(P\ x) \vee q) &\rightarrow \forall(P) \vee q \\
\forall(lx.p \vee (Q\ x)) &\rightarrow p \vee \forall(Q)
\end{aligned}$$

These examples follow our schema as well. Of course, this does not prove strong normalization, since we did not prove that the schema is compatible with such theories. The generalization is quite straightforward for commutativity but needs more investigations for more complex theories such as the above one or, associativity and commutativity together.

5 Conclusion

This paper is a continuation of [28]. Our most important contributions are the following:

(1) Our new General Schema is strong enough so as to capture strictly positive recursors, such as the recursor for Brouwer's ordinals, without compromising the essential properties of the calculus. The strong normalization proof for this extension is again based on the Tait and Girard's computability predicates technique and uses in an essential way the strict-positivity condition of the inductive types.
(2) The new formulation of the schema makes it very easy to define new extensions, by simply adding new cases to the definition of "accessibility", or new computability preserving operations in the "computable closure".
(3) The notion of "computable closure" is an important concept which has already be used in a different context [29].
(4) Several precise conjectures have been stated. The most important two, in our view, are the use of the General Schema to prove the strong normalization of higher-order rewriting *à la* Nipkow on the practical side, and the generalization of the schema to capture (non-strictly) positive inductive types on the theoretical one. The first conjecture has been recently solved by the first author in [5].

Another kind of extension should now be considered, by considering a richer type system, which we did in [6], keeping the same definition for the rules and the General Schema. But a richer type system allows us to have richer forms of rewrite rules: the General Schema should therefore be adapted so as to allow for rules of a dependent type and even rules over types. Experience shows that the latter kind of extension raises important technical difficulties. Strong elimination rules in the Calculus of Inductive Constructions [51] or the

rules defining a system of Natural Deduction Modulo [17,18] are of that kind.

References

[1] F. Barbanera and M. Fernández. Combining first and higher order rewrite systems with type assignment systems. In *Proc. of the 1st Int. Conf. on Typed Lambda Calculi and Applications*, LNCS 664, 1993.

[2] F. Barbanera and M. Fernández. Modularity of termination and confluence in combinations of rewrite systems with λ_ω. In *Proc. of the 20th Int. Colloq. on Automata, Languages, and Programming*, LNCS 700, 1993.

[3] F. Barbanera, M. Fernández, and H. 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.

[4] 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.

[5] F. Blanqui. Termination and confluence of higher-order rewrite systems. In *Proc. of the 11th Int. Conf. on Rewriting Techniques and Applications*, 2000. To appear in LNCS. Available at `http://www.lri.fr/~blanqui/`.

[6] F. Blanqui, J.-P. Jouannaud, and M. Okada. The Calculus of Algebraic Constructions. In *Proc. of the 10th Int. Conf. on Rewriting Techniques and Applications*, LNCS 1631, 1999.

[7] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236, 1999.

[8] V. Breazu-Tannen. Combining algebra and higher-order types. In *Proc. of the 3rd Symp. on Logic in Computer Science*, IEEE Computer Society, 1988.

[9] V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic strong normalization. In *Proc. of the 16th Int. Colloq. on Automata, Languages, and Programming*, LNCS 372, 1989.

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

[11] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available at `http://www.grappa.univ-lille3.fr/tata/`, 1997.

[12] T. Coquand. Pattern matching with dependent types. In *Proc. of the 3rd Work. on Types for Proofs and Programs*, Chalmers University of Technology, Sweden, 1992.

[13] T. Coquand and G. Huet. Constructions: A higher order proof system for mechanizing mathematics. In *Proc. of the 1985 European Conf. on Computer Algebra*, LNCS 203.

[14] T. Coquand and C. Paulin-Mohring. Inductively defined types. In *Proc. of the 1988 Int. Conf. on Computer Logic*, LNCS 417.

[15] N. de Bruijn. The mathematical language Automath, its usage, and some of its extensions. In *Proc. of the Symp. on Automatic Demonstration*, LNCS 125, 1970. Reprinted in: Selected Papers on Automath, edited by R.P. Nederpelt, J.H. Geuvers and R.C. de Vrijer, Studies in Logic, vol. 133. North-Holland, 1994.

[16] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 6: Rewrite Systems. North-Holland, 1990.

[17] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. Technical Report 3400, INRIA, France, 1998.

[18] G. Dowek and B. Werner. Proof normalization modulo. Technical Report 3542, INRIA, France, 1998.

[19] J. Gallier. On Girard's "Candidats de Réductibilité". In P.-G. Odifreddi, editor, *Logic and Computer Science*. North-Holland, 1990.

[20] E. Giménez. Codifying guarded definitions with recursion schemes. In *Proc. of the 5th Work. on Types for Proofs and Programs*, LNCS 996, 1994.

[21] E. Giménez. Structural recursive definitions in type theory. In *Proc. of the 25th Int. Colloq. on Automata, Languages, and Programming*, LNCS 1443, 1998.

[22] J.-Y. Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proc. of the 2nd Scandinavian Logic Symp.*, volume 63 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1971.

[23] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmetique d'ordre supérieur*. PhD thesis, Université Paris VII, France, 1972.

[24] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1988.

[25] K. Gödel. On intuitionistic arithmetic and number theory. In M. Davis, editor, *The undecidable*. Raven Press, 1965.

[26] K. Hasebe. On extensions of Gödel's System T. Master's thesis, Keio University, Japan, 2000. In japanese.

[27] J.-P. Jouannaud and M. Okada. Executable higher-order algebraic specification languages. In *Proc. of the 6th Symp. on Logic in Computer Science*, IEEE Computer Society, 1991.

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

[29] J.-P. Jouannaud and A. Rubio. The Higher-Order Recursive Path Ordering. In *Proc. of the 14th Symp. on Logic in Computer Science*, IEEE Computer Society, 1999.

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

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

[32] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Proceedings of the 73' Logic Colloquium*, volume 80 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1975.

[33] P. Martin-Löf. *Intuitionistic type theory*. Bibliopolis, 1984.

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

[35] N. P. Mendler. First- and second-order lambda calculi with recursive types. Technical Report TR 86-764, Cornell University, United States, 1986.

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

[37] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In *Proc. of the 1989 Int. Work. on Extensions of Logic Programming*, LNCS 475.

[38] B. Monate. Automates de formes normales et réductibilité inductive. Master's thesis, Université Paris-Sud, France, 1997.

[39] T. Nipkow. Higher-order critical pairs. In *Proc. of the 6th Symp. on Logic in Computer Science*, IEEE Computer Society, 1991.

[40] M. Okada. Strong normalizability for the combined system of the typed lambda calculus and an arbitrary convergent term rewrite system. In *Proc. of the 1989 Int. Symp. on Symbolic and Algebraic Computation*, ACM Press.

[41] M. Okada and P. J. Scott. A note on rewriting theory for uniqueness of iteration. *Theory and Applications of Categories*, 6(4), 2000.

[42] C. Prehofer. *Solving Higher-Order Equations: From Logic to Programming*. PhD thesis, Technische Universität München, Germany, 1995.

[43] Z. Qian. Linear unification of higher-order patterns. In *Proc. of the 7th Int. Joint Conf. CAAP/FASE on Theory and Practice of Software Development*, LNCS 668, 1993.

[44] M. P. A. Sellink. Verifying process algebra proofs in type theory. In *Proc. of the Int. Work. on Semantics of Specification Languages*, Workshops in Computing, 1993.

[45] S. Stenlund. *Combinators, Lambda-Terms and Proof Theory.* D. Reidel, 1972.

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

[47] Y. Toyama. Counterexamples to terminating for the direct sum of term rewriting systems. *Information Processing Letters*, 25(3), 1986.

[48] J. van de Pol. Termination proofs for higher-order rewrite systems. In *Proc. of the 1st Int. Work. on Higher-Order Algebra, Logic and Term Rewriting*, LNCS 816, 1993.

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

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

[51] B. Werner. *Une Théorie des Constructions Inductives.* PhD thesis, Université Paris VII, France, 1994.