Abstract data type systems

Jean-Pierre Jouannaud\textsuperscript{a}, Mitsuhiro Okada\textsuperscript{b,}\textsuperscript{*}

\textsuperscript{a}LRI, Bat. 490, CNRS/Université de Paris Sud, 91405 Orsay, France
\textsuperscript{b}Department of Philosophy, Keio University, 108 Minato-ku, Tokyo, Japan

Abstract

This paper is concerned with the foundations of an extension of pure type systems by abstract data types, hence the name of \textit{Abstract Data Type Systems}. ADTS generalize inductive types as they are defined in the calculus of constructions, by providing definitions of functions by pattern matching on the one hand, and relations among constructors of the inductive type on the other. It also generalizes the first-order framework of abstract data types by providing function types and higher-order equations. The first half of the paper describes the framework of ADTS, while the second half investigates cases where ADTS are strongly normalizing. This is shown to be the case for the polymorphic lambda calculus (with possibly subtypes) enriched by higher-order algebraic rules obeying a strong generalization of primitive recursion of higher type that we call the \textit{general schema}. This covers in particular the case of inductive types whose constructors do not have functional arguments. We conjecture that this result holds true for all calculi of the so-called Barendregt's cube. On the other hand, the definition of a schema for the higher-order rules allowing for more general inductive types is left open.

1. Introduction

Computer scientists have come up with two quite different notions of types for programming languages.

\textit{Abstract data types} aim at specifying software by encapsulating data defined abstractly by means of constructors and operations specified by a set of (first-order) directed equations operating on the constructor expressions. In this setting, computations proceed by rewriting, that is by repeatedly replacing a left hand side of equation

\footnote{Corresponding author. E-mail: mitsu@abelard.rsch.mita.keio.ac.jp.}

\footnote{This work was partly supported by the "Gr"ec de programmation du CNRS", the ESPRIT working group COMPASS, the ESPRIT working group CCL, NSERC of Canada and FCAR of Quebec, Grants-in-Aid for Scientific Research of Ministry of Education, Science and Culture of Japan, International Collaborative Research Grant of Ministry of Education, Science and Culture of Japan, the Oogata-kenkyu-Jyosei grant of Keio University, and Japan Society of the Promotion of Science. A preliminary version of this work was published in the proceeding of the International IEEE Conference on Logic in Computer Science, Amsterdam, 1991.}

0304-3975/97/$17.00 \textcopyright 1997 Published by Elsevier Science B.V. All rights reserved

PII S0304-3975(96)00161-2
by its corresponding right hand side. The search for left hand sides involves a pattern matching algorithm, hence this style of definitions is often referred to as **pattern matching definitions**. Typical of this simple approach is the abstract data type \( \text{Nat} \) given below in an OBJ-like syntax \([21, 29, 25]\), which specifies natural numbers with addition represented in Peano notation:

\[
\text{OBJ Nat} \\
\text{constructors} \\
\phantom{\text{OBJ Nat}} 0 : \text{Nat} \\
\phantom{\text{OBJ Nat}} \text{succ} : \text{Nat} \rightarrow \text{Nat} \\
\text{operators} \quad + : \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \\
\text{variables} \quad x, y : \text{Nat} \\
\text{equations} \\
\phantom{\text{OBJ Nat}} 0 + x = x \\
\phantom{\text{OBJ Nat}} \text{succ}(x) + y = \text{succ}(x + y) \\
\text{end OBJ}
\]

Following the usual jargon of first-order languages such as OBJ, \( \text{Nat} \) is called a **sort** rather than a type, 0 and \( \text{succ} \) are the **constructors**, \( + \) is an **operator** defined by the **equations**. The user can query the specification by asking for the value of an expression, say \( \text{succ}(0) + y \). This expression is first **type-checked** with respect to the operator declarations used as a bottom-up tree automaton which verifies here that the expression has sort \( \text{Nat} \). This automaton is usually described by the following typing rule, where sorts are interpreted as states:

\[
\frac{f : \sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma \quad \Gamma \vdash t_1 : \sigma_1 \ldots \Gamma \vdash t_n : \sigma_n}{\Gamma \vdash f(t_1, \ldots, t_n) : \sigma}
\]

Then, the expression is normalized by applying first the second, then the first equation, resulting in the expression \( \text{succ}(y) \) of sort \( \text{Nat} \). Note that rewriting an expression does not change its type, a property usually called **subject reduction**. It is important to note here that there is only one type in our example, the sort \( \text{Nat} \), and that only the well-formed terms have a type. For example, although there is a declaration for \( + \), \( + \) is not a term, therefore it has no type. Indeed, the language used being first-order, the operations are not data, hence need not be typed.

In this specification, \( \text{Nat} \) is usually called a **sort** rather than a type in the framework of first order language such as OBJ, 0 and \( \text{succ} \) are the **constructors**, \( + \) is an **operator** defined by the **equations**. The user can query the specification by asking for the value of an expression, say \( \text{succ}(0) + y \). This expression is first **type-checked** with respect to the operator declarations used as a bottom-up tree automaton which verifies here that the expression has sort \( \text{Nat} \). Then, it is normalized by applying first the second, then the first equation, resulting in the expression \( \text{succ}(y) \) of sort \( \text{Nat} \). Note that rewriting an expression does not change its type, a property usually called **subject reduction**. It is important to note here that there is only one type in our example, the sort \( \text{Nat} \), and that only the well-formed terms have a type. For example, although there is a
declaration for $\mathbf{+}$, $\mathbf{+}$ is not a term, hence it has no type. Indeed, the language used here being first-order, the operations are not data, hence need not be typed.

Following the ADJ group,\(^2\) we adopt the point of view that the semantics of an abstract data type is given by the (unique up to isomorphism) initial algebra in the class of algebras that satisfy the equations [24]. For most purposes, this algebra can be represented by an additional second-order axiom expressing the induction principle over the constructors (on the standard model of second-order logic):

$$\text{Ind} \triangleq \forall P. P(0) \rightarrow \forall x. (P(x) \rightarrow P(\text{succ}(x))) \rightarrow \forall y. P(y)$$

When the underlying language is first order (as is OBJ) the above second-order axiom is interpreted via the restricted comprehension principle (often called a predicative comprehension principle) where the second-order quantifications are interpreted in first-order definable sets; this restricted form of the second-order induction axiom is usually expressed as a first-order axiom scheme. Because neither one is easy to implement in an automated theorem prover, proof techniques have been developed to reduce inductive proofs to consistency proofs. These techniques avoid an explicit use of the induction axiom by replacing some restricted (but often practically meaningful) induction proofs by a feasible rewrite-based computation. They are known under the name of inductionless induction, or proof by lack of consistency [27,31].

**Functional types** propose a completely different view in which expressions are typed according to their syntactic structure by using a type constructor, the arrow $\rightarrow$. In this setting, the constant $0$ of the example above would have type $\text{Nat}$, while the (higher-order) constants $\text{succ}$ and $\mathbf{+}$ would have the respective types $\text{Nat} \rightarrow \text{Nat}$ for $\text{succ}$, and $\text{Nat} \rightarrow (\text{Nat} \rightarrow \text{Nat})$ for $\mathbf{+}$. This means that $\mathbf{+}$ is a function which, applied to an argument $a$ of type $\text{Nat}$, returns a function of type $\text{Nat} \rightarrow \text{Nat}$ whose value for input $y$ of type $\text{Nat}$ is equal to $a + y$ of type $\text{Nat}$. So, both $\mathbf{+}$ and $\mathbf{(a \mathbf{+} y)}$ as well as $\mathbf{(a \mathbf{+} y)}$ are expressions of the functional calculus. Note again that computations do not change types. Originating from Church’s simply typed lambda calculus, this notion of type fits perfectly with lambda abstraction and function composition, hence with usual functional languages, as exemplified by the two typing rules below:

\[\frac{\Sigma \vdash M : \sigma \rightarrow \tau \quad \Sigma \vdash N : \sigma}{\Sigma \vdash (MN) : \tau}\quad \frac{\Sigma \vdash (\lambda x : \sigma . M) : \sigma \rightarrow \tau}{\Sigma \vdash \{x : \sigma\} \vdash M : \tau}\]

The true understanding of functional types refers to the so-called **Curry–Howard isomorphism** (see, e.g., [6]), in which types become propositions of intuitionistic logic, while functional programs of a given type are identified with proofs of the corresponding proposition. Furthermore, the type checking rules of the programming language can be interpreted as natural deduction rules for the intuitionistic logic and the functional language can indeed be itself viewed as a proof development system for the natural deduction rules of the intuitionistic logic. Finally, computations in the functional pro-

---

\(^2\)ADJ was the acronym used by Goguen, Thatcher, Wagner and Wright in the seventies, while working out the foundations of algebraic semantics at IBM Yorktown Heights.
gram are further identified with normalization of proofs in the intuitionistic logic, and the so-called strong normalization property of the language becomes nothing but the so-called cut elimination theorem of the logic. In this identification, our basic model for typed functional languages, the simply typed lambda calculus, is identified with the intuitionistic natural deduction system restricted to propositional implication.

This logical system is of course very weak, and indeed, Church's simply typed lambda calculus has severe limitations. First, it is an extremely low-level functional language, since it lacks facilities for expressing data that must therefore be encoded by lambda terms. For an example, the natural number \( n \) can be encoded by the Church numeral \( \lambda x f. f^n x \). Second, it is an extremely poor functional language from the computational point of view. For example, the only functions operating on Church numerals that can be represented in this system are known to be the polynomial functions with test to zero. The idea of adding an abstract data type to Church's simply typed lambda calculus will remedy both problems. In the obtained Gödel's system \( T \), natural numbers are represented in the notation of Peano Arithmetic by the constructors 0 and \( \text{succ} \). From the point of view of the Curry–Howard isomorphism, the induction axiom (schema) which we introduced above to characterise the domain of the abstract data type \( \text{Nat} \), corresponds exactly to a new functional constant \( \text{rec} \) of higher type \( \text{Ind} \), called a recursor in the context of Gödel's system \( T \). Moreover, Gödel's definition of the recursor via primitive recursive rewrite rules of higher type corresponds to Gentzen's cut-elimination for Peano arithmetic: \(^3\)

\[
\begin{align*}
(\text{rec} P t u 0) & \rightarrow t \\
(\text{rec} P t u \text{succ}(x)) & \rightarrow (u x(\text{rec} P t u x))
\end{align*}
\]

The introduction of these higher-order rewrite rules based on the constructors of the abstract data type \( \text{Nat} \) leads to a very rich and neat calculus: all primitive recursive functions on natural numbers can be represented in \( T \), since a primitive recursive function definition is a special instance of the above recursor schema. Of course, this schema is more powerful than the usual schema for primitive recursive functions, since it can operate on arbitrary (possibly functional) types. For example, it is well-known that the Ackermann's function can be represented in \( T \), and this is actually true of all recursive functions which are provably total in Peano arithmetic (the so-called \( \leq \aleph_0 \)-transfinite primitive recursive functions). This is true of those provably total in second-order Peano Arithmetic for the polymorphic version of \( T \), the so-called system \( F \) of Girard.

So, Gödel's system \( T \) is a good start for integrating algebraic and functional types, and indeed, it has been generalized in at least three ways.

\(^{3}\) \( \text{Ind} \) is not a type, of course, in system \( T \), but it is in richer type systems such as Martin Löf's intuitionistic theory of types, or Girard's system \( F^{\omega} \).

\(^{4}\) To illustrate the Curry–Howard isomorphism, let us assume a richer type structure in which \( \text{rec} \) has type \( \text{Ind} \). Functional application corresponding to \( \rightarrow \)-elimination (that is, modus ponens), \( t \) and \( (\text{rec} P t u 0) \) are two terms (that is, proofs) of the same type (proposition) \( P(0) \), and \( (\text{rec} P t u \text{succ}(x)) \) and \( (u x(\text{rec} P t u x)) \) are two terms (proofs) of the same type (proposition) \( P(\text{succ}(x)) \). The corresponding rules will therefore allow to eliminate the constant \( \text{rec} \).
The first generalization aimed at extending Girard's type system [23]. A powerful
notion of quantification over types leads to even stronger facilities for constructing
types. This is the basis of the calculus of constructions of Coquand and Huet. This
calculus has several interesting subcalculi for which quantification over types is syntact-
ically restricted, yielding the simply typed lambda calculus of Church, the polymorphic
lambda calculus of Girard, the lambda calculus with dependent types of De Bruijn. The
kind of quantification, or equivalently, of impredicative comprehension principle used
by each of them, allows to organize these calculi in the so-called lambda cube of
(pure\textsuperscript{5}) type systems [6].

A consequence of the above statements (in particular due to the impredicativity of
the comprehension principles, or quantification rules) is that the addition of Gődel's
recursors does not add any more expressivity to the calculus of constructions. How-
ever, it adds simplicity; for example, as we already pointed out, the representation of
natural numbers in Peano Arithmetic is far more easy to use than the representation by
Church numerals. More generally, it is easier to define a data structure by means of
constructors, following the algebraic specifications style, than via a coding by lambda
terms. This idea is exploited by Coquand and Paulin-Mohring in the calculus of induct-
ive constructions, in which recursors are added for all inductive types by following
the Curry-Howard principle as above. The (quite complex) strong normalization proof
of this calculus was recently worked out by Werner [47].

Gődel's system $T$ as well as the calculus of inductive constructions allow the spec-
ification of initial algebras by means of recursors. But they do not allow the definition
of functions by pattern matching, as the definition of $+$ in our starting example. Nor
do they allow the specification of quotient algebras by means of equations among con-
structors, which is often needed, as in the case, for example, of bags. So, abstract data
types in their full generality, including algebraic rewrite rules, are not available in $T$
or the calculus of inductive constructions. The addition of first-order rewrite rules to
the typed lambda calculus was initiated by Breazu-Tannen [8] for the study of the
confluence property, and followed by Breazu-Tannen and Gallier [9] and Okada [40]
independently, for the strong normalization property. But these were not true general-
izations of $T$, since the rewrite rules for Gődel's recursor are higher order. The addition
of the rules for $T$ in this setting was first considered by Dershowitz and Okada in [16].
But the first true generalization of $T$ in this direction was obtained by Jouannaud and
Okada, who introduced a generalization of primitive recursion for arbitrary algebraic
types [32], which we call here the multiset recursive schema. We will indeed develop
in this paper a complete although comprehensive strong normalization proof of an even
more general calculus. This first generalization was in turn generalized by Barbanera,
Fernández and Geuvers in a series of papers, to more powerful type systems includ-
ing the calculus of constructions [2-4]. Again, the strong normalization proof becomes
quite involved when the type system is rich enough to interact with the rewrite rules.

\footnote{There are of course other pure type systems.}
Summarizing, our goal in this paper is to design and study a framework integrating typed lambda calculi with algebraic definitions of data types, including possible subtypes, which we call Abstract Data Type Systems. In particular, definitions by pattern matching are going to be available in this framework, as well as recursors and lambda definitions. We think that such a framework will provide a better theoretical foundation for both functional languages allowing pattern matching definitions like ML [39], and abstract data types with higher-order definitions. It will also yield an improved setting for developing natural deduction proof systems by providing with data types including pattern matching definitions. There is a danger, though, that such a rich framework can have inconsistent instances, and we think that this is the main reason why it did not surface before. Therefore, it is crucial to prove its logical consistency. As usual, this is done via the proof of three main properties: subject reduction, strong normalization, and confluence. We already pointed out that strong normalization is difficult. Confluence is in general much easier, once strong normalization is proved. Subject reduction is easy for the weaker calculi, but hard for the calculus of constructions. We will not face this problem here, by sticking to the polymorphic lambda calculus.

The main contribution of this paper is a strong normalization proof for abstract data type systems satisfying the two properties that the first-order definitions are terminating and conservative (variables may not have more occurrences in the right-hand side of a rule than in its left-hand one, or must otherwise be shared), and the higher-order ones follow our general recursive schema, which is a (powerful) generalization of the primitive recursive schema of higher types. This proof is carried out in various type systems, simple types, polymorphic types, subtypes, and their combinations. The proof is based on the reducibility predicate method of Tait and Girard, although it does not refer to a particular predicate, but rather to the properties that the predicate should satisfy. We of course exhibit such a predicate, which we think is interesting on its own. Finally, we also give a confluence result for which we assume further restrictions on the rules, thus ensuring consistency in this more restricted setting.

There are still quite a few pending open problems that we would like to be solved. First, the generalization of our results to more powerful type systems. This was done for the multiset schema [4], and should not be difficult to generalize to the more general schema introduced here. Second, and this is more important, our schema does not allow for recursors of data types whose constructors admit functional arguments. An example is given in conclusion, borrowed from [47]. Third, higher-order rules on types are not considered. Fourth, and this is important as well, the case of quotient types causes difficulties in providing with a general definition of recursors.

The paper is organized as follows. Section 2 gives the necessary notations and definitions about terms and rewrite rules. Section 3 presents the notion of abstract data type system. Section 4 investigates various calculi for which abstract data type systems satisfy the two properties that the first-order definitions are terminating and conservative (variables may not have more occurrences in the right-hand side of a rule than in its left-hand one, or must otherwise be shared), and the higher-order ones follow our general recursive schema, which is a (powerful) generalization of the primitive recursive schema of higher types. This proof is carried out in various type systems, simple types, polymorphic types, subtypes, and their combinations. The proof is based on the reducibility predicate method of Tait and Girard, although it does not refer to a particular predicate, but rather to the properties that the predicate should satisfy. We of course exhibit such a predicate, which we think is interesting on its own. Finally, we also give a confluence result for which we assume further restrictions on the rules, thus ensuring consistency in this more restricted setting.

There are still quite a few pending open problems that we would like to be solved. First, the generalization of our results to more powerful type systems. This was done for the multiset schema [4], and should not be difficult to generalize to the more general schema introduced here. Second, and this is more important, our schema does not allow for recursors of data types whose constructors admit functional arguments. An example is given in conclusion, borrowed from [47]. Third, higher-order rules on types are not considered. Fourth, and this is important as well, the case of quotient types causes difficulties in providing with a general definition of recursors.

The paper is organized as follows. Section 2 gives the necessary notations and definitions about terms and rewrite rules. Section 3 presents the notion of abstract data type system. Section 4 investigates various calculi for which abstract data type systems

---

6 This question has now been settled by Ralph Matthes (personal communication).
define strongly normalizing computations, including a calculus with first-order rewrite rules in Section 4.3, a polymorphic calculus in Section 4.4 with possibly subtypes in Section 4.5, and a calculus with higher-order rewrite rules obeying the so-called general recursive schema in Section 4.6. Section 5 investigates quickly the problem of confluence of the calculus. In conclusion, we present a few examples and discuss the significance and limitations of our results.

2. Preliminaries

We expect the reader to be familiar with the basic concepts and notations of term rewriting systems and typed lambda calculi. We refer to [14] for definitions and notations of term rewriting, and to [5, 6] for the notations of lambda calculi. When notations differ, we will in general favour [14].

A signature $\mathcal{F}$ is a finite set of function symbols together with their (fixed) arity. $\mathcal{X}$ denotes a denumerable set of variables, $T(\mathcal{F})$ denotes the set of ground terms over $\mathcal{F}$ and $T(\mathcal{F}, \mathcal{X})$ denotes the set of terms built up from $\mathcal{F}$ and $\mathcal{X}$. Terms are identified with finite labelled trees as usual. Positions are strings of positive integers. A denotes the empty string (root position) and "." denotes string concatenation. We use Pos$(t)$ for the set of positions in $t$, and $\mathcal{F}$Pos$(t)$ for its set of non-variable positions. The prefix ordering (resp. lexicographic ordering) on positions is denoted by $\succ$ (resp. $\succ^\text{lex}$) and the strict subterm relationship by $\ll$. The encompassment ordering, denoted by $\ll$, is the strict part of the quasi-ordering: $u \ll v$ if $v|_p = u \sigma$ for some position $p$ and substitution $\sigma$ and its equivalence corresponds to variable renaming. Subterm is a special case of encompassment, as well as subsumption for which $u$ is an instance of $v$. The subterm of $t$ at position $p$ is denoted by $t|_p$ and the result of replacing $t|_p$ with $u$ at position $p$ in $t$ is denoted by $t[u]_p$. This notation is also used to indicate that $u$ is a subterm of $t$. Var$(t)$ denotes the set of variables appearing in $t$. A term is linear if variables in Var$(t)$ occur at most once in $t$, and ground if Var$(t) = \emptyset$.

$\lambda$-terms will be considered as particular terms, therefore allowing us to reuse the same notations: for each variable $x \in \mathcal{X}$, $\lambda x.$ is a unary prefix symbol, while the hidden application operator (also denoted by $@$ when necessary) is a binary infix symbol. We use Var$(t)$ and $\exists \lambda$Var$(t)$ for, respectively, the set of free variables and the set of bound variables of $t$. Remember that we can always rename bound variables by $\alpha$-conversion in order to keep both sets disjoint. Terms over the infinite signature $\mathcal{F} \cup \{\lambda, x., @\}$ are called algebraic $\lambda$-terms, of which usual terms as well as $\lambda$-terms are particular cases.

Substitutions are written as in $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ where the $x_i$ are supposed to be all different, and $t_i$ is assumed different from $x_i$. We use greek letters for substitutions and postfix notation for their application. Remember that substitutions behave as endomorphisms defined on free variables.

A term rewriting system is a set of rewrite rules $R = \{l_i \rightarrow r_i\}$, where $l_i \notin \mathcal{X}$ and Var$(r_i) \subseteq$ Var$(l_i)$.
A term $t$ rewrites to a term $u$ at position $p$ with the rule $l \rightarrow r$ and the substitution $\sigma$, written $t \xrightarrow{p}{\sigma} u$, or simply $t \xrightarrow{R} u$, if $t|_p = l\sigma$ and $u = t[r\sigma]_p$. Such a term $t$ is called reducible. Irreducible terms are said in normal form. A term $t$ is strongly normalizable if every reduction sequence out of $t$ is finite, hence ends in a normal form of $t$. A substitution $\gamma$ is strongly normalizable if $x\gamma$ is strongly normalizable for all $x$. We denote by $\rightarrow^+_R$ (resp. $\rightarrow^*_R$) the transitive (resp. transitive and reflexive) closure of the rewrite relation $\rightarrow_R$, and by $s \Downarrow^+_R t$ the joinability relation, that is $s \rightarrow^+_R u$ and $t \rightarrow^*_R u$ for some $u$. The subindex $R$ will be omitted when clear from the context. A term rewriting system $R$ is
- confluent if $t \rightarrow^* u$ and $t \rightarrow^* v$ implies $u \rightarrow^* s$ and $v \rightarrow^* s$ for some $s$,
- terminating (or strongly normalizing) if all reduction sequences are finite,
- convergent if it is confluent and terminating.

We sometimes speak of a strongly normalizing, or confluent, or convergent relation on a subset of the whole set of terms. This assumes of course that this subset is closed under rewriting.

We will make intensive use of well-founded orderings for proving strong normalization properties. In particular, the following results will play a key role, see [14]:

Assume $\rightarrow$ is a terminating rewrite relation. Then $\rightarrow \cup \succ$ is well-founded.

Assume $\rightarrow_1$ and $\rightarrow_2$ are well-founded orderings on sets $S_1, S_2$. Then $(\rightarrow_1, \rightarrow_2)_{lex}$ is a well-founded ordering on $S_1 \times S_2$.

Assume $\succ$ is a well-founded ordering on a set $S$. Then $\succ_{mul}$ is a well-founded ordering on the set of multisets of elements of $S$. This ordering is defined as the transitive closure of the following relation on multisets (using $\cup$ for multisets union):

$$M \cup \{s\} \succ M \cup \{t_1, \ldots, t_n\} \quad \text{if} \quad s > t_i \quad \forall i \in [1..n]$$

3. Abstract data type systems

Our purpose in this section is to introduce precisely the kind of combined language we are going to investigate. To this end, we define first types, then terms. For simplicity, we consider only one type operator, namely $\rightarrow$, although our results accommodate other type operators as well, e.g., product types and sum types. We will introduce two different ways of building terms, with and without Currying, and discuss their respective merits. The section will culminate with a tentative definition of what we really mean by an abstract data type system, and what are the important properties of these type systems.

3.1. The language

We start with types, usually called sorts for algebraic terms and types for lambda-terms, and continue with terms before to give the typing rules.

Sorts: We are first given a set of sort operators of a given arity, $S = \bigcup_{n \geq 0} S_n$ where $S_n$ is the set of sort operators of arity $n$, and a set of (first-order) sort variables
The set of sorts is the term-algebra $\mathcal{A}(\mathcal{Y}, \Psi_1)$, and we use $s$ and $t$ to denote arbitrary sorts. The set of sorts is equipped with a rewrite ordering $\succ_u$. Hence, if $s \succ_u s'$, then $t[s] \succ_u t[s']$. In practice, the subsort relationship is a finite bottom-up tree automaton [11], that is the rewrite relation generated by pairs $s \succ_u t$ standing for $s(\xi_1, \ldots, \xi_m) \succ_u t(\xi_1, \ldots, \xi_n)$ where $s$ and $t$ are sort functions such that $m \leq n$, and $\xi_1, \ldots, \xi_n$ are appropriate distinct sort variables. Note that sort operators are enough already to have polymorphic data types.

**Types:** We now define more general types needed for $\lambda$-terms. Let $\Psi$ be a denumerable set of type variables containing $\Psi_1$. The set $\mathcal{F}_\Psi$ of types is defined recursively by the following context-free grammar, where $\mathcal{F}_\sigma$ and $\Psi$ are considered non-terminals generating the elements in their respective sets,

$$\mathcal{F}_\Psi := \Psi \mid \mathcal{F}_\Psi \mathcal{F}_\Psi \mid (\mathcal{F}_\Psi \rightarrow \mathcal{F}_\Psi) \mid (\forall \Psi. \mathcal{F}_\Psi)$$

In the following, we use $\psi$ and $\xi$ to denote type variables, and $\sigma$ and $\tau$ to denote types. We will also use $\sigma_1 \rightarrow (\sigma_2 \rightarrow \cdots (\sigma_n \rightarrow \sigma) \cdots)$ as an abbreviation for $\sigma_1 \times \sigma_2 \times \cdots \times \sigma_n \rightarrow \sigma$. We do not assume that $\sigma$ is a basic type in these expressions, whose use is explained below.

Closed-type expressions are categorized according to which type operator they are headed by. **Functional types** have the form $\sigma \rightarrow \tau$ where $\sigma$ and $\tau$ are types, while **quantified types** have the form $\forall \xi. \sigma$ where $\sigma$ is a type, and $\xi$ is a type variable. **Basic types** are sort expressions.

**Terms:** We now assume given a set of function symbols which are traditionally seen as higher-order constants by Currying, i.e., $\mathcal{F} = \bigcup_{\sigma_1, \ldots, \sigma_n, \sigma} \mathcal{F}_{\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma}$, where $\mathcal{F}_\tau$ denotes the set of function symbols of type $\tau$. Since these function symbols are meant to be algebraic operators, we will actually adopt a slightly different notation for their type, by using $\sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma$ instead of $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma$. Hence, $\mathcal{F} = \bigcup_{\sigma_1, \ldots, \sigma_n, \sigma} \mathcal{F}_{\sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma}$. Although we could explicitly introduce product types in our system, we will instead consider the former writing as an other form of the latter (a kind of abbreviation), which is simply meant to record that the function symbol $f$ of type $\sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma$ takes as inputs exactly $n$ arguments of respective types $\sigma_1, \ldots, \sigma_n$ and outputs a result of (possibly functional) type $\sigma$. $n$ is called the **arity** of $f$, and is defined by the declaration that $f \in \mathcal{F}_{\sigma_1 \times \cdots \times \sigma_n \times \sigma}$. While the output type $\sigma$ may always be considered as a basic type when function symbols are seen as higher-order constants, this is no more the case when algebraic symbols must come with all their arguments at once. This is why the declaration must specify the output type.

These two different views of function symbols seen either as Curried (higher-order) constants to be applied to their arguments sequentially, or as algebraic functions to be applied to all their input arguments at once, will yield two different views of terms in the sequel.

The result of a function $f$ may of course be of functional type, as well as its arguments. In case $\sigma_1, \ldots, \sigma_n$ and $\sigma$ are all basic types, then $\mathcal{F}_{\sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma}$ is called the set of **first-order function symbols** of type $\sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma$. We denote by $\mathcal{F}_1$ the set of all first-order function symbols and by $f, g$ its elements. We use $F, G$ to denote
higher-order function symbols, that is function symbols the arguments or the result of which are not all of basic type.

For example, let Nat and List() be sort operators. \( +: \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \) is a first order function symbol, while \( \text{map}:(\text{Nat} \rightarrow \text{Nat}) \times \text{List(Nat)} \rightarrow \text{List(Nat)} \) is a higher-order function symbol.

Many algebraic specification languages, e.g. OBJ [21], tolerate overloading: a function symbol may have different arities. This causes some difficulties in presence of subtypes which can be resolved under the condition that two copies of the same symbol agree on the intersection of their respective types [26]. We prefer however to rule out overloading in order to avoid technical difficulties which are not relevant for our purpose.

The traditional way to build lambda terms with algebraic symbols is by considering the Curried function symbols as constants in the calculus. This was the approach taken by Breazu-Tannen and his followers. The set of Curried \( \lambda \)-terms is then defined recursively by the following context-free grammar, where \( \mathcal{X} \) denotes the set of variable symbols, containing a subset \( \mathcal{X}_1 \) of first-order variables:

\[
\mathcal{F} := \mathcal{F} \mid \mathcal{X} \mid (\mathcal{F} \mathcal{F}) \mid (\lambda \mathcal{X} : \mathcal{X}_g.\mathcal{F}) \mid (\mathcal{F}_g \mathcal{F}) \mid (\Lambda \Psi.\mathcal{F})
\]

The application of a term to a type is called type application. The abstraction over a type variable in a term is called type abstraction. Type abstraction may be bounded as in \( (\Lambda \Psi \subseteq \mathcal{X}_g.\mathcal{F}) \), meaning that the abstracted type variable must be smaller (in a sense to be made precise later) than some given type. We denote by \( \mathcal{F}(\mathcal{F},\mathcal{X}) \) the subset of (higher-order) algebraic terms, and by \( \mathcal{F}(\mathcal{F}_1,\mathcal{X}_1) \) its subset of first-order algebraic terms. We use \( x, y, z \) to denote first-order variables, \( X, Y, Z \) for higher-order ones and \( u, v, w, l, r \) as well as \( M, N \) for arbitrary terms.

As already noticed, this notion of term does not follow the tradition of algebraic specifications. We will therefore adopt a second notion of term, which does not allow the algebraic symbols to be terms of the calculus. In this setting, algebraic symbols will come with all their arguments inside parentheses, hence they are not Curried constants. For the moment, we will not require the presence of the right number of arguments for the algebraic symbols, this will be taken care of later by the typing rules. In this second presentation, the set of algebraic \( \lambda \)-terms, is defined by the following grammar:

\[
\mathcal{G} := \mathcal{G} \mid (\mathcal{G} \mathcal{G}) \mid \mathcal{G}(\mathcal{G}, \ldots, \mathcal{G}) \mid (\lambda \mathcal{X} : \mathcal{X}_g.\mathcal{G}) \mid (\mathcal{G}_g \mathcal{G}) \mid (\Lambda \Psi.\mathcal{G})
\]

Note that function symbols occur now at non-leaf positions in terms while variables occur at leaf positions only (as well as constants). This is not the case with Curried \( \lambda \)-terms for which the internal nodes are labelled by lambdas and applications only.

These two ways of building terms are reflected in the typing rules to come. However, we can immediately remark that every algebraic \( \lambda \)-term is a particular Curried \( \lambda \)-term, obtained by dropping the superfluous commas and parentheses. Conversely, to every Curried \( \lambda \)-term corresponds an algebraic \( \lambda \)-term, called its \( \eta \)-expanded form, obtained (roughly speaking) by adding the necessary parentheses, commas and possibly \( \lambda \)-binders. For example, the Curried \( \lambda \)-term \( + \) corresponds to the algebraic \( \lambda \)-term \( \lambda x \lambda y.(x + y) \).
So, both calculi will be strongly related, although they will not have exactly the same properties. We will later formalize a useful relationship between both calculi.

3.2. Typing rules

As usual, typing rules allow to restrict the set of terms by constraining them to follow a precise discipline. Below, we give the most general set of typing rules that can reasonably be dealt with. In each section to come, we will make clear which subset of these rules is used.

We assume given a type assignment $\Gamma$. In our general setting, a type assignment comes in two parts: a set of pairs of the form $x : \sigma$ associating the type $\sigma$ to the (free) variable $x$; a set of pairs of the form $\xi \subseteq \sigma$ stating that the type variable $\xi$ is a subtype of the type $\sigma$.

Our typing judgements are written as $\Gamma \vdash M : \sigma$ if the term $M$ can be proved to have the type $\sigma$ under the hypotheses in $\Gamma$, or $\Gamma \vdash \sigma \subseteq \tau$ if the type $\sigma$ can be proved to be a subtype of $\tau$ under the hypotheses in $\Gamma$. The rules below typecheck Curried $\lambda$-terms:

**Variables:** $x : \sigma \in \Gamma \\
\Gamma \vdash x : \sigma$

**Application:** $\Gamma \vdash M : \sigma \rightarrow \Gamma \vdash N : \sigma \\
\Gamma \vdash (MN) : \tau$

**Type application:** $\Gamma \vdash M : \forall \xi \phi \cdot \sigma \Gamma \vdash \tau \subseteq \phi \\
\Gamma \vdash (\forall \xi \phi \cdot M) : \sigma \rightarrow \tau$

**Bounded type application:** $\Gamma \vdash (\forall \xi \phi \cdot M) : \sigma \rightarrow \tau \\
\Gamma \vdash (\forall \xi \phi \cdot \psi) \cdot \sigma \Gamma \vdash \tau \subseteq \psi \\
\Gamma \vdash (\forall \xi \phi \cdot \psi) \cdot \sigma \rightarrow \tau$

**Subsorts:** $(\alpha, \tau) \subseteq (\beta, \phi) \\
\Gamma \vdash \alpha \subseteq \beta \\
\Gamma \vdash \beta \subseteq \phi \\
\Gamma \vdash \alpha \subseteq \phi$

**Rewrite:** $\Gamma \vdash g \Theta \in \alpha \\
\Gamma \vdash [g \Theta] \in \alpha$

where $\gamma$ is a sort expression $\Theta$ is a substitution

**Restriction:** $\Gamma \vdash \delta \subseteq \gamma \Gamma \vdash \sigma \subseteq \tau \\
\Gamma \vdash \gamma \rightarrow \sigma \subseteq \tau$

Type checking the set of algebraic $\lambda$-terms requires the single change of the rule **Functions** above by the following new version:

**Functions:** $f \in \mathcal{F}_{\sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma} \\
\Gamma \vdash t_1 : \sigma_1 \ldots \Gamma \vdash t_n : \sigma_n \\
\Gamma \vdash f(t_1, \ldots, t_n) : \sigma$

Note that this rule checks the number $n$ of arguments of $f \in \mathcal{F}_{\sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma}$.

We say that the term $M$ has type $\sigma$ in the environment $\Gamma$ if $\Gamma \vdash M : \sigma$ is provable in the above inference system. We say that a term $M$ is typable in the environment $\Gamma$ if there exist a type $\sigma$ such that $M$ has type $\sigma$ in the environment $\Gamma$. A term $M$ is typable if it is typable in some environment $\Gamma$. Computing an environment $\Gamma$ in which a given term $M$ is typable is called the type reconstruction problem. In practice,
we are not interested in all possible environments in which $M$ is typable, but in minimal ones with respect to the subsumption ordering, called principal types.

Type inference as well as type checking are undecidable in the previous type system, even without type quantification, due to the combination of higher-order types with subtypes. Type inference becomes decidable for simple type disciplines (no functional types). Principal types are then unique (up to renaming of variables), provided the subsort structure has good properties [28]. Type checking is of course an easier task, allowing for functional types and inheritance together. See [45] for a survey of type inference problems.

In the sequel, we sometimes use a subset of the above rules only. More precisely, for our strong normalization argument of Section 3, we consider the following sublanguages: monomorphically typed lambda calculus (cf. Section 4.3), corresponds to the rules Variables, Functions, Application, Abstraction; the polymorphically typed lambda calculus considered in Section 4.4 needs in addition the rules Type application and Type abstraction; the rules handling subtypes will be used in Section 4.5. Section 4.6 assumes the rule Functions for algebraic functions of higher type.

3.3. Computation rules

Definition 1. A (higher-order) equation $I = r$ is a non-oriented pair of algebraic $\lambda$-terms having the same type. The equation is first-order if so are $I$ and $r$, that is $I, r \in \mathcal{T}(\mathcal{F}_1, \mathcal{X}_1)$, and higher-order otherwise.

A (higher-order) rule $l \rightarrow r$ is an oriented pair of algebraic $\lambda$-terms such that the type of $r$ is a subtype of the type of $l$ and $\forall \text{var}(r) \subseteq \forall \text{var}(l)$. A rule is first-order if so are $l$ and $r$. We will use $R$ for an arbitrary set of first-order rules, and $\rightarrow_R$ for the associated rewrite relation. A higher-order rule $\rightarrow r$ is algebraic if the root of $l$ is an algebraic function symbol. We will denote by $\text{HOR}$ an arbitrary set of algebraic higher-order rules possibly containing first-order ones, and by $\rightarrow_{\text{HOR}}$ the associated rewrite relation.

Note that we exclude rules in which some subterm would not be an algebraic $\lambda$-term (but would be a curried term). We do not think that this is a real restriction for practice, and has the advantage that the set of algebraic $\lambda$-terms is closed under reduction.

We keep the notations above for rewriting the algebraic part of an arbitrary term, and denote by $\rightarrow_{\text{mix}}$ the following rewrite relation:

$$\rightarrow_{\text{mix}} = \rightarrow_R \cup \rightarrow_{\text{HOR}} \cup \rightarrow_\beta \cup \rightarrow_{\eta} \cup \rightarrow_{T_\varepsilon, \beta} \cup \rightarrow_\eta \cup \rightarrow_{T_\eta} \cup \rightarrow_{T_\varepsilon \eta}$$

where classically:

$$(\lambda x : \sigma. u) v \beta \rightarrow_{\beta} u \{ x \mapsto v \}$$

$$\lambda x : \sigma. u x \eta \rightarrow_{\eta} u, \text{ if } x \notin \mathcal{F} \forall \text{var}(u)$$
We also use $\Rightarrow_{\text{mix}}$ for the least congruence on terms generated by the associated rewrite relation. A key property of $\Rightarrow_{\text{mix}}$ is the following.

**Lemma 2.** (Subject reduction). Let $u \Rightarrow_{\text{mix}} v$ and $\Gamma \vdash u : \sigma$. Then $\Gamma \vdash v : \tau$ for some $\tau$ such that $\tau \leq \sigma$.

In other words, under our assumptions, all rewrite relations are type decreasing. In case there are no subsorts, the relations are type preserving. We do not give a proof of this classical result.

We are now in a position to explain the main differences between algebraic $\lambda$-terms and Curried $\lambda$-terms. Assume $f$ is a function symbol in $\mathcal{F}_{\sigma \rightarrow \sigma}$, where $\sigma$ is a sort. Consider the set of first-order rewrite rules

$$R = \{ f(x) \rightarrow x \}.$$

We have the following non-confluent diagram for Curried $\lambda$-terms:

$$f \eta \quad \lambda x.f \ x \rightarrow \lambda x.x$$

This happens of course because $f$ is a term in the calculus. If we choose the other alternative, by considering algebraic $\lambda$-terms, the problem disappears. It does not mean that we cannot use the function of name $f$ when considering algebraic $\lambda$-terms, but that we need to express it differently, by using the term $\lambda x.f(x)$ instead. So, $\eta$-expansions are implicitly used for algebraic symbols in the setting of algebraic $\lambda$-terms, while $\eta$-reductions can still be used for the other terms. The conclusion is that confluence will require using algebraic $\lambda$-terms when considering the $\eta$-rule.

The situation is quite different with strong normalization. Since the set of algebraic $\lambda$-terms is naturally embedded into the set of Curried $\lambda$-terms and is closed under reduction, it will be enough to prove strong normalization for Curried $\lambda$-terms. However, it will turn out that Curried $\lambda$-terms introduce many technical difficulties. We will therefore encode them by algebraic $\lambda$-terms of an enlarged signature, as explained now.

Given the original signature $\mathcal{F}$, we define a new signature $\mathcal{G}$ which includes for each operator symbol $f \in \mathcal{F}_{\sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma}$, $n + 1$ operators $f^i \in \mathcal{G}_{\sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma_{i-1} \times \cdots \times \sigma_{n-i} \times \sigma}$ for $i \in [0..n]$. Note that $i$ is the number of arguments expected by $f^i$. We can now canonically interpret a Curried $\lambda$-term $u$ over the signature $\mathcal{F}$ as an algebraic term $u$ over the signature $\mathcal{G}$ by replacing each higher-order constant $f \in \mathcal{F}_{\sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma}$ by the constant $f^0$ of the same type. In addition, we also consider the following set $\text{Alg}$ of (non-algebraic) higher-order rewrite rules operating on algebraic $\lambda$-terms over the
signature $\mathcal{G}$:

$$\text{Alg} = \{(f^{i-1}(t_1, \ldots, t_{i-1}, t_i) \rightarrow f^i(t_1, \ldots, t_{i-1}, t_i) \forall f \in \mathcal{F}_{\sigma_1 \times \ldots \times \sigma_n} \rightarrow \sigma \forall i \in [1..n]}\}$$

The rules in $R$ and $\text{HOR}$ use the formalism of algebraic $\lambda$-terms as stated in our definition of a rule. We actually can as well interpret an algebraic $\lambda$-term $u$ over the signature $\mathcal{F}$ as an algebraic $\lambda$-term $u$ over the signature $\mathcal{G}$ by replacing each occurrence of an algebraic symbol $f$ of arity $n$ by $f^n$ (thus identifying $f$ with $f^n$, something we will do without notice in the sequel). Let us call $R_{\text{alg}}$ and $\text{HOR}_{\text{alg}}$ the obtained sets of rules. We now consider the following relation on algebraic $\lambda$-terms built over $\mathcal{G}$:

$$\rightarrow_{\text{mixalg}} = \rightarrow_{R_{\text{alg}}} \cup \rightarrow_{\text{HOR}_{\text{alg}}} \cup \rightarrow_{\text{Alg}} \cup \rightarrow_{\beta} \cup \rightarrow_{\tau_{\beta}} \cup \rightarrow_{\eta} \cup \rightarrow_{\tau_{\epsilon}}$$

The next lemma relates these constructions.

**Lemma 3.** For all Curried $\lambda$-terms $u, v \in \mathcal{F}(\mathcal{F}, \mathcal{F})$ $u \rightarrow_{\text{mix}} v$ implies $u \rightarrow_{+_{\text{mixalg}}}^+ v$.

Being straightforward, the proof of this lemma is left to the reader. This lemma will allow us to prove strong normalization results for Curried $\lambda$-terms with the advantage of considering algebraic $\lambda$-terms instead, and the disadvantage of having additional non-algebraic higher-order rules to consider.

Note finally that we can now interpret the counterexample to confluence of the $\eta$-rule for Curried $\lambda$-terms inside the calculus on algebraic $\lambda$-terms over the extended signature $\mathcal{G}$:

$$f^0 \xrightarrow{\eta} \lambda.x.f^0 x \xrightarrow{\text{Alg}} \lambda.x.f^1(x) \xrightarrow{R} \lambda.x.x$$

While the original non-confluent critical pair involved the $\eta$-rule with a rule of $R$, it now involves the $\eta$-rule with a rule of $\text{Alg}$.

We will prove strong normalization results for a suitable subset of the above rules in several consecutive subsections of Section 4, namely:

- Section 4.3 (algebraic $\lambda$-terms): $\rightarrow_{\text{mix}} = \rightarrow_{R} \cup \rightarrow_{\beta} \cup \rightarrow_{\eta}$
- Section 4.4 (algebraic $\lambda$-terms): $\rightarrow_{\text{mix}} = \rightarrow_{R} \cup \rightarrow_{\beta} \cup \rightarrow_{\tau_{\beta}} \cup \rightarrow_{\eta} \cup \rightarrow_{\tau_{\epsilon}}$
- Section 4.5 (algebraic $\lambda$-terms): $\rightarrow_{\text{mix}} = \rightarrow_{R} \cup \rightarrow_{\beta} \cup \rightarrow_{\tau_{\epsilon}} \cup \rightarrow_{\eta} \cup \rightarrow_{\tau_{\epsilon}}$
- Section 4.6 (Curried $\lambda$-terms): $\rightarrow_{\text{mix}} = \rightarrow_{R_{\text{alg}}} \cup \rightarrow_{\beta} \cup \rightarrow_{\eta} \cup \rightarrow_{\text{HOR}_{\text{alg}}} \cup \rightarrow_{\text{Alg}}$

So, we consider the strong normalization property of algebraic $\lambda$-terms in all sections but Section 4.6, in which we consider Curried $\lambda$-terms instead. We will however always state our results for Curried $\lambda$-terms, because the (more complex) induction argument used in the latter section applies as well in the former ones. The reason for this choice is that we want to make the first strong normalization proof in Section 4.3 as simple as possible by getting rid of the additional difficulties introduced by Curied $\lambda$-terms. We will also simplify our notations in Section 4.6 by using $\rightarrow_{\text{mix}}$ with the meaning of $\rightarrow_{\text{mixalg}}$, and also by dropping the subscript $\text{alg}$ from $R_{\text{alg}}$ and $\text{HOR}_{\text{alg}}$, and the
superscript \( n \) from the algebraic symbols of arity \( n \). We will also sometimes drop the subscript \( \text{mix} \) by using \( \rightarrow \) for \( \rightarrow_{\text{mix}} \). Note finally that we will not develop the case of polymorphism and subtypes in connection with higher-order rules. The arguments are basically the same as for the case of first-order rules, and they are already developed in Sections 4.4 and 4.5.

3.4. Abstract data type systems

Abstract data type systems are abstract data types whose operators may be defined by a confluent and terminating set of rewrite rules on algebraic \( \lambda \)-terms. The definition is therefore parameterized by the type system in use. More precisely, an abstract data type system comes with the following ingredients:

- A set \( \Sigma \) of sorts, called basic types (or type constructors),
- For each sort \( \sigma \in \Sigma \), a set of constructors \( \mathcal{C}_\sigma \) defining the set of constructor expressions \( \mathcal{T}(\bigcup_{\sigma \in \Sigma} \mathcal{C}_\sigma) \), together with a (possibly empty) first-order rewrite system \( S \) assumed to be convergent on ground constructor terms,
- For each sort \( \sigma \in \Sigma \), a set of typed operators together with, for each such operator \( f \):
  1. If \( f \) is a first-order symbol, a set of first-order rewrite rules \( R_f \) of the form 
     \[ f(t_1, \ldots, t_n) \rightarrow t, \] 
     where \( t_1, \ldots, t_n, t \) are arbitrary algebraic terms, such that \( f(u_1, \ldots, u_n) \) is reducible for arbitrary ground constructor terms \( u_1, \ldots, u_n \).
  2. If \( f \) is a higher-order symbol, a set of higher-order rewrite rules \( \text{HOR} \), including the rules for Gödel’s recursors, of the form 
     \[ f(t_1, \ldots, t_n) \rightarrow t, \] 
     where \( t_1, \ldots, t_n, t \) are arbitrary algebraic \( \lambda \)-terms, such that \( f(u_1, \ldots, u_n) \) is reducible for arbitrary ground constructor \( \lambda \)-terms \( u_1, \ldots, u_n \).
- Subject reduction, confluence and strong normalization properties for \( \rightarrow_{\text{mix}} \).

Before justifying the properties required from an abstract data type system, let us first illustrate the definition with the example of polymorphic lists, the syntax following the OBJ style:

OBJ List[\( x \)]

constructors

\( \text{nil} : \text{List}(x) \)

\( \text{cons} : x \times \text{List}(x) \rightarrow \text{List}(x) \)

operators

\( \text{rec} : (x \times \text{List}(x) \times \text{List}(x) \rightarrow \text{List}(x)) \times \text{List}(x) \times \text{List}(x) \rightarrow \text{List}(x) \)

\( \text{Last} : \text{List}(x) \rightarrow x \)

\( \text{Reverse} : \text{List}(x) \rightarrow \text{List}(x) \)

variables

\( x : x \)

\( l : \text{List}(x) \)

\( l' : \text{List}(x) \)

\( X : x \times \text{List}(x) \times \text{List}(x) \rightarrow \text{List}(x) \)

equations
\[
\text{rec}(X, l', \text{nil}) = l' \\
\text{rec}(X, l', \text{cons}(x, l)) = X(x, l, l', \text{rec}(X, l', l)) \\
\text{Last}(l) = \text{rec}(\lambda x. l \cdot l' \cdot l'' \cdot l''); l, l) \\
\text{Reverse}(l) = \text{rec}(\lambda x. l \cdot l' \cdot l'' \cdot \text{cons}(x, l''), \text{ nil}, l)
\]

end OBJ

The main problem with algebraic definitions is whether they are hierarchically consistent, that is, whether the definitions may equate constructor terms which are not equal in the theory generated by the equations on constructors [48].

**Definition 4.** An abstract data type system is said to be *hierarchically consistent* iff for any two ground constructor terms \(s\) and \(t\) such that \(s =_{\text{mix}} t\), then \(s =_{=} t\).

**Proposition 5.** Abstract data type systems are hierarchically consistent.

**Proof.** By confluence of \(\rightarrow_{\text{mix}}\), we deduce that \(s \rightarrow_{\text{mix}}^{*} u \leftarrow_{\text{mix}}^{*} t\). Now, since \(s\) and \(t\) are built from constructor symbols, only the rules in \(S\) may apply.

Another important property of definitions is whether functions are completely defined, that is, all cases are really covered. If this is the case, then any closed term will eventually rewrite to a constructor term.

**Definition 6.** An abstract data type system is said to be *complete* iff for any closed term \(s\) there exist a constructor term \(t\) such that \(s =_{\text{mix}} t\).

Unfortunately, the hypotheses above are not sufficient for completeness, because a closed term \(s\) may have subterms with free variables headed by operators. However, a weaker property is of course true, namely, if a term \(s\) has the property that any one of its subterms is closed whenever it is headed by an operator, then \(s\) rewrites via \(\rightarrow_{\text{mix}}\) to a term with the same property, and we can conclude by noetherian induction on \(\rightarrow^{+}\) that it will eventually reduce to a constructor term. So, we introduce a weaker notion of completeness that is satisfied by abstract data type systems.

**Definition 7.** An abstract data type system is said to be *algebraically complete* iff for any closed term \(s\) whose all subterms headed by operators are also closed, there exist a constructor term \(t\) such that \(s =_{\text{mix}} t\).

**Proposition 8.** Abstract data type systems are algebraically complete.

One may of course question whether the properties required for algebraic completeness are decidable. It turns out that the condition given for the first-order operators is decidable [44, 10]. For higher-order rewrite rules, it is of course undecidable in general, but becomes decidable for a surprisingly large second-order fragment covering all practical cases of second-order definitions [37].
4. Strong normalization of abstract data type systems

Strong normalization of typed lambda calculi is always a difficult task. In our case, there is an additional difficulty originating from the intricate interaction of two different calculi, the lambda calculus, and the algebraic calculus. We will not illustrate it now, but rather start by explaining how strong normalization proofs will be carried out, before applying the technique to the case of first-order rules, and then only, to the case of higher-order rules. This allows us to introduce much of the apparatus needed for carrying out the most difficult case, with higher-order rules.

Following Girard [23], we reduce the strong normalization proof for various combinations between a typed lambda calculus and a terminating set of rewrite rules to an abstract form. The technique will first be presented by considering the simpler case of first-order rewrite rules. We will then reduce the polymorphic case to the monomorphic case, as indicated without an explicit proof in [40], and the case of subtypes as well. We finally show how to deal with the more complex case of higher-order rewrite rules.

4.1. Reducing strong normalization to the principal case

In contrast to the other proofs [9, 40, 41, 1, 18], our abstract proof has the following properties:
- It does not depend on a particular version of the typed lambda calculus, whose strong normalization proof, though, must be carried out by Tait–Girard’s computability predicate method.
- It does not depend on a particular choice of the computability predicate: the proof of the main lemma does not refer to the predicate itself.
- The strong normalization property of the abstract data type system is reduced to a property called Principal Case which does not involve the computability predicate.
- The addition of the second-order polymorphism to the underlying lambda calculus does not really modify the strong normalization proof of the whole language: we again reduce the polymorphic case to the monomorphic case, hence to the Principal Case.
- Adding suitable higher-order rewrite rules simply needs a (non-trivial) modification of the Principal Case.
- Other properties of the language, e.g. weak normalization, could be proved within the same schema.

We first recall the familiar Tait–Girard’s computability predicate method [23]. One first inductively defines a computability predicate \( R_\tau \) for each type \( \tau \). We say that \( u \) is a computable term of type \( \tau \) if \( R_\tau(u) \) is true. Let \( P \) be the property associated to the computability predicate. We prove that the property \( P \) is true for any term of any type. This is usually done in two steps:

*Step 1:* If a term is computable then it has property \( P \). This is normally proved by induction on types.
Step 2: Every term is computable. This is normally proved by structural induction on terms.

As easily seen from this abstract structure of the proof, one need not to fix a specific property for $P$, such as strong normalizability, as usually done in the literature. One need not either work with a specific predicate, as usually done also. In order to specify the properties that the computability predicate should satisfy, we need the notion of a neutral term. Girard calls neutral a term $u$ such that the rule used for typing $u$ at the root is not an introduction rule. The only type operator we explicitly mention in this paper is $\lambda$, but there may be others, such as pairing, sums, etc. We could consider them to be the price of a (small) expansion factor in the proofs. So, non-neutral terms should therefore be abstractions. This will be the case in the setting of algebraic $\lambda$-terms, but there will be additional non-neutral terms in the setting of Curried $\lambda$-terms, associated to the higher-order functional constants. Our definition applies to both settings.

Definition 9. A term is neutral if its application to any other term does not rewrite at the root.

This definition of a neutral term is not standard, and is indeed directly related to the presence of Curried algebraic symbols. This is why it does not correspond to other definitions in the literature. It is however crucial for obtaining the required properties for the computability predicate.

Neutral (and non-neutral) terms enjoy the following characteristic properties.

Lemma 10. (i) terms of base type are neutral.

(ii) the set of non-neutral terms is closed under reduction.

(iii) Abstractions are the only non-neutral terms in the setting of algebraic $\lambda$-terms.

(iv) Abstractions and terms of the form $f^j(t_1, \ldots, t_j)$ are the only non-neutral terms in the setting of Curried $\lambda$-terms.

We can now give the properties needed for $P$. The first two express relationships between the computability property and the property $P$, while the last three express closure properties of the computability property.

C1. If $s$ is computable, then $s$ has property $P$.

C2. If $s$ is of base type and satisfies the property $P$, then $s$ is computable.

C3. If $s$ is computable and $s \rightarrow^* t$ then $t$ is computable.

C4. If $s$ is neutral and $t$ is computable for all $t$ such that $s \rightarrow t$, then $s$ is computable.

C5. If $s$ has type $\tau \rightarrow \sigma$, then $s$ is computable iff $(s \, t)$ is computable for all computable $t$ of type $\tau$.

From these computability properties, we observe that terms of base type are computable iff they are strongly normalizable. The following key properties of non-neutral terms follow also from the above properties.
Lemma 11. Assume that the terms \( v, t \) and \( v\{x \mapsto t\} \) are computable. Then \( (\lambda x.v)t \) is computable.

Proof. Since \( v \) and \( t \) are computable, they are strongly normalizable by C1. The proof is by induction on the pair \((v, t)\) ordered by \( (\longrightarrow^+_{\text{mix}})_{\text{mul}} \) and use of property C4. Since \( v\{x \mapsto t\} \) is computable by assumption, we simply need to consider rewrites inside \( \lambda x.v \) and \( t \). If \( v \longrightarrow_{\text{mix}} v' \), then \( v\{x \mapsto t\} \longrightarrow_{\text{mix}} v'\{x \mapsto t\} \), hence both \( v' \) and \( v'\{x \mapsto t\} \) are computable by C3, and \( (\lambda x.v)t \) is therefore computable by using the induction hypothesis. The case where \( t \longrightarrow_{\text{mix}} t' \) is similar.

Lemma 12. Let \( j < \text{arity}(f) \) and assume that the terms \( t_1, \ldots, t_{j-1} \) and \( f^{j+1}(t_1, \ldots, t_{j+1}) \) are computable. Then \( (f_j(t_1, \ldots, t_j)t_{j+1}) \) is computable.

Proof. The proof is basically the same as above, by induction on the multiset \( \{t_1, \ldots, t_j\} \) of strongly normalizable terms ordered by \( (\longrightarrow^+_{\text{mix}})_{\text{mul}} \).

There are many different computability predicates defined in the literature for the strong normalization property, some of which are collected in [22]. Some other definitions which can easily handle strong normalization with product and coproduct types may be found in [42]. The most important observation to construct our abstract proof of strong normalization is that all known definitions of a computability predicate for the strong normalization property satisfy the computability properties listed above. So, defining a particular predicate is not really needed, except for proving that it satisfies all properties. This is shown by induction on the inductive definition of the computability predicate when the property is not trivially true.

4.2. A computability predicate for strong normalizability

As a simple example, we consider the following definition of a computability predicate \( R \) for the strong normalization property \( P \) of the algebraic language based on the simply typed \( \lambda \)-calculus:

1. Let \( u \) be a non-neutral term of type \( \tau \rightarrow \sigma \). \( R_{\tau \rightarrow \sigma}(u) \) iff \( R_{\sigma}(u \ t) \) for all computable \( t \) of type \( \tau \).

2. Let \( u \) be a neutral term of type \( \tau \). \( R_\tau(u) \) if \( \forall v \) such that \( u \longrightarrow_{\text{mix}} v \), then \( R\_\tau(v) \).

Our predicate is defined by induction on types on the one hand (for non-neutral terms), and for each type \( \tau \), by recursion (for neutral terms) on the other. In order to guarantee the existence of the computability predicates, we need to show that each predicate \( R_\tau \) has the least fixpoint. This is done by induction on types, by showing that for each type \( \tau \) the underlying functional is monotonic on the lattice of subsets of the set of terms ordered by set inclusion. We therefore invoke Tarski’s theorem on the existence of the least fix point. As a consequence, we can prove properties of the predicate by a double induction operating lexicographically on types first, and then on the recursion level. In the sequel, we will word it as an induction on the definition of the predicate.
The above predicate applies to any calculus over algebraic or Curried \( \lambda \)-terms as introduced in Section 3. In particular, it applies to the simply typed lambda calculus by taking \( \to_{\beta \eta} \) for \( \to_{\text{mix}} \).

In the subsequent subsections, we investigate more and more complicated calculi, mixing typed lambda-calculi with first-order rewriting first, then with first and higher-order rewriting, as already mentioned in Section 3.3. We will not prove the computability properties again and again, but only for the case of the simply typed lambda calculus with first and higher-order rules, which is more general than needed for the next section, and can be easily generalized for the more powerful type systems as sketched in the appropriate subsections. Thanks to our abstract definition of a non-neutral term, our proofs do not need to refer to a particular one of our two settings, Curried \( \lambda \)-terms via their encoding by algebraic \( \lambda \)-terms in the extended signature defined in Section 3.3, or algebraic \( \lambda \)-terms over the user’s signature.

We now start proving the computability properties.

**Property 1.** (C1) *Computable terms are strongly normalizable.*

**Proof.** The proof proceeds by induction on the definition of the computability predicate. We distinguish two cases, according to the definition of the predicate.

- Let \( u \) be a non-neutral computable term of type \( \tau \to \sigma \). Since \( x \in \mathcal{X}_\tau \) is neutral and strongly normalizable, then \( R_\tau(x) \) holds by case 2 of the definition of predicate. Hence, \((u \, x)\) is computable by definition of the predicate, and strongly normalizable by the induction hypothesis. Hence, its subterm \( u \) is strongly normalizable as well.

- Let \( s \) be neutral. By definition of the predicate, every \( t \) such that \( s \to_{\text{mix}} t \) is computable, hence strongly normalizable by the induction hypothesis, which implies that \( s \) is strongly normalizable.

**Property 2.** (C2) *Strongly normalizable terms of base type are computable.*

**Proof.** Let \( u \) be a strongly normalizable term of base type \( \tau \). The proof is by induction on the set of strongly normalizable terms ordered by \( \to^{+}_{\text{mix}} \). All terms \( v \) such that \( u \to_{\text{mix}} v \) are strongly normalizable and of base type \( \tau \), hence they are computable by induction hypothesis. Since base type terms are neutral, \( u \) is therefore computable by the definition of the predicate.

**Property 3.** (C3) *If \( u \) is computable, and \( u \xrightarrow{*_{\text{mix}}} v \), then \( v \) is computable.*

**Proof.** The proof of the property in case of a single reduction step proceeds by case analysis. The property then follows by a straightforward induction on the length of the derivation. Again, we distinguish two cases:

- Let \( u \) be a non-neutral computable term of type \( \tau \to \sigma \) and let \( u \to_{\text{mix}} v \), hence \( v \) is non-neutral as well. To show that it is computable, we therefore need to show that \((v \, t)\) is computable for all computable terms \( t \) of type \( \tau \). Since \( u \) is computable,
by definition of the predicate, \((u \ t)\) must be computable for all computable terms \(t\) of type \(\tau\). Since \((u \ t)\) is neutral, and \((u \ t) \mix (v \ t)\), then \((v \ t)\) must itself be computable and we are done.

- Let \(u\) be neutral of type \(\tau\) and \(u \xrightarrow{\text{mix}} u'\) for some \(u'\) of type \(\tau\). Then \(R_r(u)\) implies that for all \(v\) such that \(u \xrightarrow{\text{mix}} v\), \(R_r(v)\) holds. In particular, \(R_r(u')\) holds.

**Property 4.** (C4) If \(s\) is neutral and \(t\) is computable for all \(t\) such that \(s \rightarrow t\), then \(s\) is computable.

This property is actually built in our definition of the predicate.

**Property 5.** (C5) Let \(u : \tau \rightarrow \sigma\). Then \(u\) is computable iff \((u \ t)\) is computable for all computable \(t : \tau\).

**Proof.** The proof is again by induction on the definition of the predicate, distinguishing 2 cases:

- Let \(u\) be a computable non-neutral term of type \(\tau \rightarrow \sigma\). This case follows from the definition of the predicate.

- Let \(u\) be neutral of type \(\tau \rightarrow \sigma\), hence \((u \ t)\) is neutral of type \(\sigma\) for all computable \(t\) of type \(\tau\). Assume that \((u \ t)\) is computable for all computable \(t\). Since \(u\) is neutral, we simply need to show that \(u'\) is computable for all \(u'\) such that \(u \xrightarrow{\text{mix}} u'\). Since \((u \ t)\) is also neutral, then \((u' \ t)\) is computable by definition of the predicate. Hence, \(u'\) is computable by induction hypothesis, and we are done. Assume that \(u\) is computable. We need to show that \((u \ t)\) is computable for all computable terms \(t : \tau\). Since \((u \ t)\) is neutral, this reduces to the computability of its reducts. But \(t\) is strongly normalizable by (C1), hence we can restrict ourselves to normalized terms \(t\). By Lemma 10, the only possible reduct is of the form \((u' \ t)\), where \(u'\) is a reduct of \(u\), hence is computable by (C3). Hence, \((u' \ t)\) is computable by induction hypothesis and we are done.

Note that all above the proofs rely on properties of reductions. A different property \(P\) would require different proofs. They also rely on the particular definition of the predicate, since we use its definition for building the inductions.

### 4.3. Strong normalization of an algebraic functional language

We now start investigating the case of an algebraic functional language obtained by adding first-order rewrite rules to the typed lambda calculus. We will consider the case of algebraic \(\lambda\)-terms. We could as well consider the case of Curried \(\lambda\)-terms via Lemma 3, this will actually be done in Section 4.6. We stick here to this case in order to show the basic ingredients of our proof technique in a simple case.

Let us give the precise statement for step 2. We say that the substitution \(\gamma = \{x_1 : \sigma_1 \mapsto u_1, \ldots, x_n : \sigma_n \mapsto u_n\}\) is computable if \(R_{\sigma_i}(u_i)\) holds true for all \(i \in [1..n]\). By definition, we will consider that the substitution \(\gamma\) is defined for the variables in \(\forall \text{var}(u)\) exactly.
Lemma 13 (First-order monomorphic case). For any term $u[x_1, \ldots, x_n]$ of type $\sigma$, whose free variables are $x_1 : \sigma_1, \ldots, x_n : \sigma_n$, if $\gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n \mapsto u_u\}$ is computable, then $w = u_\gamma$ is computable.

In Tait–Girard's method, this lemma is proved by structural induction on $u$. We can follow a similar argument, except when the root symbol of $u$ is some $f \in \mathcal{F}_1$. In this case, we will refer to an additional lemma, called Principal Case. We now proceed with the proof. The key point is that it does not need to refer to a particular computability predicate, provided it satisfies the computability properties. Nor does it refer to a particular property $P$, provided the principal case holds true.

Proof. We interpret the term $w = u_\gamma$ by the pair $\langle u, \{\gamma\} \rangle$, where $\{\gamma\}$ is the multiset $\{x_\gamma \mid x \in \text{Var}(u)\}$. These pairs are compared in the ordering $(\cdot \triangleright \cdot)$, denoted by $\triangleright$ in the following and by $\triangleright$ to indicate which pair has strictly decreased in the ordering. Here, $\triangleright$ stands for the well-founded encompassment ordering, that is $u_\triangleright v$ if $u|_p = v_\gamma$ for some position $p \in \text{Pos}(u)$ and substitution $\gamma$. Note that encompassment contains strict subterm. Besides, since $\triangleright_{\text{mix}}$ is well-founded on $\gamma$ by assumption, and because the union of the strict subterm relationship with a rewrite relation is well-founded [14], the relation $(\cdot \triangleright \cdot)_{\text{mut}}$ is well-founded on $\gamma$. Hence, $\triangleright$ is a well-founded ordering on the pairs.

Note that a term may have several interpretations, depending on how it is viewed as an instance of some term $t$ by a strongly normalizable substitution $\gamma$. This peculiarity will be heavily used in the proof.

We consider seven cases, depending upon the properties of $w = u_\gamma$.

1. Let $u$ be a non-neutral term, that is, an abstraction $\lambda x. v$ of type $\sigma \rightarrow \tau$ in the setting of algebraic $\lambda$-terms. To show that $w = u_\gamma$ is computable, it suffices to show that $(u_\gamma s)$ is computable for all computable terms $s$ of type $\sigma$, and use property (C5). Since $u_\gamma \triangleright v_\gamma$, $v_\gamma$ is computable by induction hypothesis. We show now that $v_\gamma\{x \mapsto s\}$ is computable. Define $v_\gamma' = v_\gamma \cup \{x \mapsto s\}$, hence $v_\gamma'$ is computable. Now, $u_\gamma \triangleright v_\gamma'$, hence $v_\gamma' = v_\gamma\{x \mapsto s\}$ is computable by induction hypothesis. We can therefore apply Lemma 11 and conclude that $(u_\gamma s)$ is computable.

2. Let $u$, hence $w$, be neutral. We prove that it is computable by using property (C4), that is for every $w'$ such that $w \triangleright_{\text{mix}} w'$, then $w'$ is computable (this also takes care of terms in normal form).

(a) Let $p \notin \mathcal{F}\text{Pos}(u)$. Note that this case takes care of the case where $u$ is a variable. Let $p = q p'$ for $u|_q \in \mathcal{X}$, $u' = u[z]_q$, where $z$ is a new variable of the appropriate type, and $\gamma' = \gamma \cup \{z \mapsto w'|_q\}$. Note that $w|_q \triangleright_{\text{mix}} w'|_q$. Since $w|_q \in \{\gamma\}$ is computable by assumption, $w'|_q$ is computable by property (C3), hence $\gamma'$ is computable. Now, $u = u'$ if the variable $u|_q$ has exactly one occurrence in $u$, otherwise $u \triangleright u'$. In the first case $u_\gamma \triangleright u_\gamma'$, and $u_\gamma \triangleright u_\gamma'$ in the second case. In both cases, $u_\gamma' = u_\gamma' = w'$ is computable.
(b) Let $p \in \mathcal{P}(u)$, with $p \neq A$, that is $p = i.p'$, for some $i \in N$ and $p' \in N^*$. Then $u \triangleright u|_i$, hence $u|_i \triangleright u|_i = (u|_i)|_i$, and $(u|_i)|_i$ is computable. Let $u' = u|_i$, for some new variable $z$ of the appropriate type, and $p' = z.p''$, for some $i \in N$ and $p'' \in N^*$. Hence $u|_i \triangleright u|_i = (u|_i)|_i$, and $(u|_i)|_i$ is computable. Finally, $u|_i \triangleright u|_i = w$ because bound variables in $u|_i$ are not bound outside $u|_i$, since $u$ is not an abstraction. Hence, $w$ is computable and $w'$ is computable by property (C3).

(c) Let $p = A, u = (\lambda x.v)t$ and $w = (\lambda x.v)(t')$. Hence $u \triangleright t$, and $w|_i \triangleright t|_i$ and $t|_i$ is computable. Let now $\gamma' = \gamma \cup \{z \mapsto t|_i\}$, hence $\gamma'$ is computable. Since $u \triangleright r$, we have $u|_i \triangleright v|_i = w'$ and $w'$ is computable.

(d) Let $p = A, u = y \triangleright z.r_\gamma = \lambda x.v$, hence $w = (\lambda x.v)(t')$ and $w' = v\{x \mapsto t|_i\}$. Since $u \triangleright t$, $w|_i \triangleright t|_i$, hence $t|_i$ is computable. $y|_i = \lambda x.v$ is computable by assumption on $\gamma$. Hence, $(v t|_i)$ is computable by property (C5), and $w' = v\{x \mapsto t|_i\}$ by property (C3).

(e) Let $p = A, u = f(u_1,\ldots,u_n)$ and $u$ is not a variable renaming of $v = f(x_1,\ldots,x_n)$. For $i \in [1,n]$, $u|_i \triangleright u|_i$, hence $\gamma'$ such that $x_i|_i = u_i|_i$ is computable. But $u \triangleright v$, since $u$ is not a renaming of $u$. Hence $v|_i = w$ is computable. Hence $w'$ is computable by property (C3).

(f) Let $p = A, u = f(z_1,\ldots,z_n)$ and $w = w|_i \rightarrow g w|_i$. Since $\gamma$ is computable by assumption, the terms $z_i|_i$ are strongly normalizable by property (C1) as well as their subterms. We then apply the Principal Case described next to show that $u|_i$ itself is strongly normalizable. Since $u|_i = w$ is of basic type, (C2) shows that it is computable. Hence $w'$ is computable by property (C3).

We are left with the statement and the proof of the Principal Case for the strong normalization property. To deal with terms whose head is algebraic, we will extract their algebraic cap, a very natural tool used for various modularity problems, such as the present one [9,40], and also unification problems [7].

**Definition 14.** Let $\xi$ be an arbitrary one to one mapping between a set $\mathcal{X}$ of variables and quotient of the set of equivalence classes of terms modulo $\beta$-conversion by the equality $=_\text{mix}$. The cap of a term $w$ is the algebraic term $\omega[x_1,\ldots,x_n]_{p_1,\ldots,p_n}$ such that
- $\forall i \in [1,n]$, the root of $w|_{p_i}$ is not in $\mathcal{F}_1$,
- $x_i = \xi[w_{|p_i}]$. Let alien$(w)$ be the multiset $\{w|_{p_i} : i \in [1,n]\}$ of alien subterms of $w$. The estimated cap of a term $w$, written out as ec$(w)$ is the cap of its $\beta\eta$-normal form.

**Example 1.** Let $\mathcal{F} = \{f : z \times z \times (z \rightarrow z) \rightarrow (z \rightarrow z)\}$, $\mathcal{X} = \{x,y,z : x\}$, and $s = f((\lambda x.x) z, (\lambda x.x) y, f(z, f(z, z, \lambda x.x) z, \lambda y.y)))$. Then cap$(s) = f(x_1, x_2, f(x_1, x_3, x_4))$, and alien$(s) = \{(\lambda x.x) z, (\lambda x.x) y, f(z, z, \lambda x.x) z, \lambda y.y\}$. 

Note that the estimated cap of a term is unique (up to \( \xi \)) due to the Church–Rosser property of the original functional language. Note also that \( \text{cap}(u) \rightarrow^p_R \text{cap}(v) \) whenever \( u \rightarrow^p_R v \) with \( p \in \mathcal{P}(\text{cap}(u)) \). The latter property is crucial for our proof, and requires coherent abstractions along computations, which is achieved by the mapping \( \xi \).

**Lemma 15** (Principal Case). A term whose root is algebraic is strongly normalizable whenever its alien subterms are strongly normalizable.

**Proof.** Since the original algebraic rewrite system is terminating, any reduction sequence starting from the cap must be finite. Hence, the cap can be used for building a transfinite induction: for this, we consider the pair \((\text{ec}(w), \text{alien}(w))\), with its associated lexicographic ordering \((\rightarrow^R, (\rightarrow^\text{mix}\text{mul})_{\text{lex}})\), which will be well-founded for our purpose, since \( \rightarrow^\text{mix} \) is well-founded on alien subterms by assumption. We are therefore left with the proof that our complexity measure above decreases for any kind of reduction (since we are proving strong normalization) applied to a term \( w \) whose root is algebraic.

1. If \( w \rightarrow^p_{\text{mix}} v \) with \( p \in \mathcal{P}(\text{cap}(w)) \), then \( w \rightarrow^p_R v \), and \( \text{ec}(w) \rightarrow_R \text{ec}(v) \).
2. If \( w \rightarrow^p_{\text{f}h} v \), with \( p \notin \mathcal{P}(\text{cap}(w)) \), then \( \text{ec}(w) = \text{ec}(v) \) and \( \text{alien}(w) \rightarrow^\text{mix} \text{alien}(v) \).
3. If \( w \rightarrow^p_R v \) with \( p \notin \mathcal{P}(\text{cap}(w)) \), then \( \text{alien}(w) \rightarrow^\text{mix} \text{alien}(v) \). On the other hand, \( \text{ec}(w) \rightarrow^\text{mix} \text{ec}(v) \) or \( \text{ec}(w) = \text{ec}(v) \). The inequality holds when the \( \lambda \)-normal form of the alien subterm of \( w \) reduced in the algebraic reduction performed on \( w \) is an algebraic term. The equality holds in all other cases. In both cases, the complexity has strictly decreased.

We can now state the strong normalization property of algebraic functional languages.

**Theorem 16.** Assume that \( R \) is a set of terminating first-order rules. Then \( \rightarrow^R \) is strongly normalizing on Curried (resp. algebraic) \( \lambda \)-terms.

**Proof.** Since variables are computable by property (C4), the identity substitution is computable. Hence, all algebraic \( \lambda \)-terms on the extended signature \( G \) are computable by the main lemma, hence normalizable for the relation \( \rightarrow^\text{mix} \) by property (C1). Applying Lemma 3 concludes the proof for the case of algebraic \( \lambda \)-terms. The case of Curried \( \lambda \)-terms will be treated in Section 4.6 as already mentioned.

Adding new rules in the functional language requires looking at new reduction cases in the above proof, and making sure that the arguments used are still valid or can be tuned up. However, all interesting cases, including the product-type, coproduct-type, uniqueness of product-type (surjective pair), uniqueness of coproduct-type, can be easily obtained, as usual.

Note also that our proof does not rely on a particular definition of the computability predicate for the property \( P \), nor does it rely on the particular property \( P \) itself: It essentially reduces the whole proof process to the Principal Case whose proof must
be shown for the particular property \( P \) one is interested in. Of course, proving the Principal Case for various properties \( P \) such as confluence, convergence, and weak-normalization may require different induction schemas and different arguments as well.

4.4. Reduction of the polymorphic case to the monomorphic case

We now extend the above proof to the case of polymorphic type systems. In particular, we will see why the inclusion of second order, higher order or even much stronger impredicative types does not influence the basic framework given in the above proof. The basic idea of Girard’s method is to consider the domain of the “candidates” of computability (reducibility) predicate of each second-order type, where a candidate \( \rho_\sigma \) of type \( \sigma \) is such a predicate on the terms of type \( \sigma \) which satisfies the computability properties mentioned above. Then the inductive definition of the computability predicate \( R_{\sigma[\xi]}(\rho) \) is given with respect to the second-order type \( \sigma[\xi] \) (with free type variables \( \xi \)) and with respect to the candidates \( \rho \) for \( \xi \). The previous definition is therefore relativized by the occurrences of the type variables and of the candidates, except that one needs to add the following new cases:

Let \( u \) be a term of type \( \sigma[\xi] \).

3. If \( \sigma[\xi] = \forall \psi.\tau[\xi,\psi] \) and \( u = \Lambda \psi.v[\psi] \), then, \( R_{\sigma[\xi]}(\rho)(u) \) if for any type \( \varsigma \) and for any candidate \( \sigma \) of type \( \varsigma \), \( R_{\sigma[\varsigma]}(\rho)(v[\varsigma]) \).

4. If \( \sigma[\xi] \) is a type variable \( \psi \), then \( R_{\sigma[\psi]}(\rho)(u(\langle \psi \mapsto \tau \rangle)) \).

If we restrict ourselves to first-order polymorphism by having \( \Psi = \Psi_1 \) (i.e., under the Curry–Howard isomorphism, the propositional variables range over the atomic formulae), then \( \varsigma \) in 3. and \( \tau \) in 4. belong to the base types.

Under these relativizations, (C5) still holds: for any \( u \) of a (ground) base type \( s \), \( R_{\sigma[\xi]}(\rho)(u) \) iff \( u \) is strongly normalizable. In fact, if \( \sigma \) is a (ground) base type, the computability predicate \( R_{\sigma[\varsigma]}(\rho)(u) \) can be treated as in the monomorphic case.

Lemma 4.3, of course, needs to be relativized by the addition of polymorphism.

**Lemma 17** (First-order polymorphic case). For any term \( u[\xi, y] \) of type \( \tau[\xi] \), where \( \xi \) (respectively \( y \)) is the list of free type variables (free variables of type \( \sigma[\xi] \)), if \( R_{\sigma[\xi]}(\rho)(v') \) for each \( v' \in v \) then \( R_{\tau[\sigma[\xi]]}(\rho)(w) \), where \( w = u[y, v] \).

**Proof.** The proof is again carried out by induction on the structure of \( u \), as usual. Since the other cases are not influenced by the addition of the first-order algebraic rules, we simply need to check the two cases for which the base types are involved. When \( \tau \) is a ground base type, then we can simply use (C5) to reduce this case to the Principal Case of the monomorphic case. When \( \tau \) is a type variable (either a usual type variable or a base type variable) substituted by a base type \( \sigma_k \), then the lemma trivially holds by the usual argument.

4.5. Subtypes at the functional level

With the presence of subtype-inference rules, the treatment of the monomorphic and polymorphic type-inference systems in the previous sections can be relativized to the
subtype-inference systems, in a natural way. Thanks to Lemma 2, no new problem arises, hence our results hold true in these cases too. We shall take a close look at this below.

The basic idea is interpret a computability predicate for a subtype as a subset (i.e., subpredicate) of the computability predicate for the supertype. In particular, the subtype of a polymorphic type is interpreted as a bounded quantification while the original polymorphic type is interpreted as a full quantification.

We now extend the above proof of the principal case to the cases of the monomorphic and polymorphic subtype systems. In particular, we will see why the inclusion of those subtyping inferences does not influence the basic framework given in the above proof.

We recall that our subtype-supertype structure is formed in the following manner. (1) the subtype structure of the base types (sorts) are determined by the order-sorted declaration; (2) the subtype structure on the higher types is determined by the subtyping inference rules (based on the subtype structure of the base types); and (3) the subtype structure of polymorphic types are determined by the bounded quantifier inferences (polymorphic subtype inferences), based on the monomorphic subtype structure.

As for the case for monomorphic subtyping system, the extension is straightforward. The computability predicates for a base type is defined as before, namely, as the set of strongly normalizable terms of a given base type. Then, the subtype relation of the base types corresponds to the subset (set-inclusion) relation of computability predicates (for the base types), which means the “inheritance” rule is satisfied in the base-types level. The former definition of the computability predicates (for higher types) preserves the subset relation of the subtypes, which actually satisfies the subtyping inference rules for higher types. These facts are expressed by the following modified form of the former lemma.

**Lemma 18** (First-order monomorphic case with subtyping). (1) For any assertion (of a type inference proof) \( x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash u[x_1, \ldots, x_n] : \sigma \), if \( \gamma = \{ x_1 : \sigma_1 \mapsto u_1, \ldots, x_n : \sigma_n \mapsto u_n \} \) is computable, then \( w = u\gamma \) is computable.

(2) For any assertion (of a type inference proof) \( \vdash \sigma \subseteq \tau \) of type \( \gamma \), \( R_\sigma \subseteq R_\tau \).

The proof is carried out by the induction on the number of type-inference rules (for \( u[x_1, \ldots, x_n] \) of type \( \sigma \) with a given environment). In particular, for the case where a subtyping is involved, the above set-inclusion interpretation of computability predicates works.

As for the polymorphic subtyping system, the extension does not cause any further problem. The basic idea of extending Girard’s method to the subtyping case is to consider a subdomain of the “candidates” of computability (reducibility) predicate of each second-order supertype, where a candidate \( \rho_\sigma \) of type \( \sigma \) is such a predicate on the terms of type \( \sigma \) which satisfies the computability properties mentioned above. Then the inductive definition of the computability predicate \( R_{\sigma[\xi]}(\rho) \) is given with respect to the second-order type \( \sigma[\xi] \) (with free type variables \( \xi \)) and with respect to the candidates \( \rho \) for \( \xi \). The previous definition is therefore relativized by the occurrences of the type
variables and of the candidates, except that one needs to add the following new cases: Let \( u \) be a term of type \( \sigma[\xi] \).

5. If \( \sigma[\xi] = \forall \psi \leq \phi. \tau[\xi, \psi] \) and \( u = A\psi \leq \phi. x[\psi] \), then, \( R_{\sigma[\xi](\rho)}(u) \) if for any type \( \zeta \leq \phi \) and for any candidate \( q \) of type \( \zeta \), \( R_{\tau[\xi, \zeta](\rho, \psi)}(v[\psi]) \).

Under these relativizations, Fact 1 still holds: for any \( u \) of a (ground) base type \( s \), \( R_{\tau[\xi]}(u) \) if \( u \) is strongly normalizable. In fact, if \( \sigma \) is a (ground) base type, the computability predicate \( R_{\sigma}(u) \) can be treated as in the monomorphic case.

Lemma 1.3, of course, needs to be relativized by the addition of bounded polymorphism.

Lemma 19 (First-order polymorphic case with subtyping). (1) For any assertion (of a type-inference proof) \( \gamma : \sigma[\xi], x_j \leq x_k \vdash u[\xi, y] : \tau[\xi] \), where \( \xi \) is the list of free type variables, if \( R_{\sigma[\xi](\rho)}(v') \) for each \( v' \in v \), and if \( \gamma_j \subseteq \gamma_k \), then \( R_{\tau[\xi, \xi](\rho)}(w) \), where \( w = u[\gamma, v] \).

(2) For any assertion (of a type-inference proof) \( \gamma : \sigma[\xi], x_j \leq x_k \vdash \delta[\xi] \leq \tau[\xi] \), where \( \xi \) is the list of free type variables, if \( R_{\sigma[\xi](\rho)}(v') \) for each \( v' \in v \), and if \( R_{\gamma} \subseteq R_{\rho} \), then \( R_{\tau[\xi]}(p_{\gamma}) \subseteq R_{\tau[\xi]}(p_{\rho}) \).

Proof. The proof is again carried out by induction on the number of the type inferences for \( u[\xi, y] : \tau[\xi] \) with a given environment. When \( \tau \) is of the form of bounded polymorphic type, the usual Girard’s argument works with the above new definition of the computability predicate for a bounded polymorphic type. Since the other cases are not influenced by the addition of the first-order algebraic rules, we simply need to check the two cases for which the base types are involved. But, those cases are exactly the two cases we checked for the polymorphic system in the previous section.

4.6. Higher-order rewrite rules

The addition of higher-order rewrite rules to typed lambda calculi does not result in a strongly normalizing calculus in general. This is why, for example, there are syntactic restrictions on the definition of inductive types in the calculus of constructions [43]. We first illustrate the kind of difficulties we need to face via several examples.

First, adding higher-order rewrite rules to first-order rewrite rules may already raise difficulties with termination, since the higher-order rules may interact with the first-order ones. Indeed, higher-order rewrite rules can be used to simulate first-order ones, and it is well known that termination is not, in general, a modular property of first-order rewrite systems. That is, the union of two confluent and terminating set of first-order rules may be non-terminating, even when both sets have no function symbol in common, as shown by Drosten’s example below [19], a refinement of Toyama’s classical example [46]:

\[
R_1 = \{ f(0, 1, x) \rightarrow f(x, x, x), \ f(x, y, z) \rightarrow 2, \ 0 \rightarrow 2, \ 1 \rightarrow 2 \},
\]

\[
R_2 = \{ g(x, y, y) \rightarrow x, \ g(y, y, x) \rightarrow x \}
\]
with the infinite reduction sequence.

\[
\begin{align*}
f(g(0, 2, 1), g(0, 2, 1), g(0, 2, 1)) \quad & \xrightarrow{R_1} f(g(0, 2, 2), g(0, 2, 1), g(0, 2, 1)) \\
& \xrightarrow{R_2} f(0, g(0, 2, 1), g(0, 2, 1)) \xrightarrow{R_1} f(0, g(2, 2, 1), g(0, 2, 1)) \\
& \xrightarrow{R_2} f(0, 1, g(0, 2, 1)) \xrightarrow{R_1} f(g(0, 2, 1), g(0, 2, 1), g(0, 2, 1)) \ldots
\end{align*}
\]

On the other hand, termination is a modular property of first-order conservative rewrite systems.

**Definition 20.** A rule \( l \rightarrow r \) is said to be conservative if a given variable cannot have more occurrences in \( r \) than in \( l \). A rewrite system \( R \) is conservative if all its rules are conservative.

Non-conservative rewrite systems enjoy conservative reductions by using sharing [20]: if a given variable occurs more times on a right-hand side of a rule than on its left-hand side, several of its occurrences on the right-hand side may be shared, and the above definition applies again by considering the right-hand side as a dag. So, non-conservative rewrite systems enjoy conservative reductions by sharing some variables on the right-hand sides of rules. It is known that termination is a modular property of conservative reductions of arbitrary constructor sharing rewrite systems [35,20]. We will therefore restrict our results to conservative first-order reductions (using possibly sharing for the first-order variables).

Let us now consider the next problem, the interaction between two higher-order rules yielding a non-terminating reduction. Let

\[
\begin{align*}
\text{List} : & \text{ is a sort constructor} \\
\alpha : & \text{ is a type variable} \\
\text{cons} : & \alpha \rightarrow \text{List}(\alpha) \rightarrow \text{List}(\alpha) \\
x : & \alpha \\
y : & \text{List}(\alpha) \\
X : & \text{List}(\alpha) \rightarrow \alpha \rightarrow \alpha \\
c : & (\text{List}(\alpha) \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \\
F : & \text{List}(\alpha) \rightarrow (\text{List}(\alpha) \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \\
G : & \text{List}(\alpha) \rightarrow \alpha \rightarrow \alpha \\
\text{HOR} = \{ & F(\text{cons}(\alpha, l), X) \rightarrow X(l, x), \ G(y, c(X)) \rightarrow F(\text{cons}(c(X), y), X) \}
\end{align*}
\]

We obtain the following non-terminating sequence:

\[
F(\text{cons}(c(G), l), G) \rightarrow G(l, c(G)) \rightarrow F(\text{cons}(c(G), l), G) \ldots
\]

Although we have used \( G \) as a term in this sequence, this is not essential. A similar (although slightly more complicated) non-terminating sequence can be obtained by using \( \lambda y. x. G(y, x) \) instead. We can see that the higher-order variable was used to disguise a mutual recursion between \( F \) and \( G \). In order to rule out such counterexamples, we
will consider a kind of primitive recursive schema on which higher-order variables used on the right-hand side of rules must occur as immediate subterms of the left hand side, a condition not satisfied by the second rule above. Since the type of such a variable must then be a subterm of the type of the higher-order constant being defined, these mutual recursions will be forbidden.

Finally, there may also be interactions between the higher-order rules and the typed lambda-calculus by appropriately instantiating the higher-order variables. Let

\[
\begin{align*}
  f &: \quad s \times s \to s \quad (f \text{ is a first-order function symbol})
  \\
  F &: \quad s \to s \quad (X \text{ is a higher-order variable symbol})
  \\
  x &: \quad s \quad (x \text{ is a first-order variable symbol})
  \\
  R &= \{ f(Xx,x) \to f(Xx,Xx) \} \quad (R \text{ is a higher-order rewrite system})
\end{align*}
\]

It can easily be seen that the above rule is terminating since each rewrite eliminates a redex. However, instantiating \( X \) with a \( \lambda \)-term expressing the identity function yields the non-terminating derivation

\[
f((\lambda y. y)x,x) \to_R f((\lambda y. y)x,(\lambda y. y)x) \to_\beta f((\lambda y. y)x,x) \to \cdots.
\]

Again, our schema will eliminate such interactions by making sure that the recursive calls operate on smaller arguments (with respect to the subterm ordering) than the starting call. Of course, in the non-terminating rule

\[
F(succ(x)) \to (\lambda x. F(x))(succ(x))
\]

the argument \( x \) of \( F \) in the right-hand side should not be considered smaller than its left hand side argument \( succ(x) \). This means that the notion of subterm must care of possible bound variables.

**Statuses and orderings:** Function symbols in an incremental development will be assigned a status, which will be used to build an ordering. We allow for a complex notion of status (e.g. the usual multiset status, or the left-to-right lexicographic status, or a term like \( \text{lex}(x_2,x_4,\text{mul}(x_3,x_1)) \)), allowing us to tune up the comparison of lists according to practical needs. For example, comparing the lists of terms \( l_1,l_2,l_3,l_4 \) and \( r_1,r_2,r_3,r_4 \) according to the above status and the ordering \( \triangleright \) generates the comparison of \( l_2 \) and \( r_2 \) first, then of \( l_4 \) and \( r_4 \) if \( l_2 \) and \( r_2 \) are equivalent under variable renaming, and if this is also true of \( l_4 \) and \( r_4 \) then the two multisets \( \{ l_3,l_1 \} \) and \( \{ r_3,r_1 \} \) are compared.

**Definition 21.** Let \( \text{mul} \) and \( \text{lex} \) be varyadic symbols. A *status* of arity \( n \) is a term \( \text{stat} \in \mathcal{F}(\{\text{mul},\text{lex}\}, X) \) such that \( \text{flat}(\text{stat}) = \{ x_1, \ldots, x_n \} \), which is in normal form with respect to the rewrite system \( R_{\text{stat}} \):

\[
\begin{align*}
  \text{lex}(x_1,.,.,.,x_m,\text{lex}(y_1,.,.,.,y_n),z_1,.,.,.,z_p) & \to \text{lex}(x_1,.,.,.,x_m,y_1,.,.,.,y_n,z_1,.,.,.,z_p) \\
  \text{mul}(x_1,.,.,.,x_m,\text{mul}(y_1,.,.,.,y_n),z_1,.,.,.,z_p) & \to \text{mul}(x_1,.,.,.,x_m,y_1,.,.,.,y_n,z_1,.,.,.,z_p)
\end{align*}
\]
To each function symbol $f$ of arity $n$, we associate a status (i.e. a tree) $stat_f$ of the same arity. The status is called multiset (respectively lexicographic) if equal to $\text{mul}(x_1, \ldots, x_n)$ (respectively $\text{lex}(x_{\xi(1)}, \ldots, x_{\xi(n)})$), where $\xi$ is a permutation of $[1..n]$). The status of a constant symbol is therefore equal to $\text{lex}$ or $\text{mul}$, while the status of a unary symbol is equal to the variable $x_1$.

Given a term $t$ whose variables occur at positions $p_1, \ldots, p_m$ such that $p_i < p_i + 1$, its status $stat_t$ is the normal form according to $R_{stat}$, of $stat(t, A, \{p_1, \ldots, p_m\})$, where $stat$ is defined by

$$
stat(f(t_1, \ldots, t_n), p, l) = stat\{x_1 \mapsto stat(t_1, p, l), \ldots, x_n \mapsto stat(t_n, p, n, l)\}
$$

$$
stat(x, p, l) = x, \text{ if } p = p_i \in l
$$

**Example 2.** Let $f$, $g$ and $a$ have the respective statuses $\text{lex}(x_1, x_2, x_3)$, $\text{mul}(x_1, x_2)$ and $\text{lex}$. Then, $f(x, y, g(x, y))$ has the status $\text{lex}(x_1, x_2, \text{mul}(x_3, x_4))$, while $f(x, g(a, y), g(a, a))$ has the status $\text{lex}(x_1, x_2)$.

Statuses allow us to extend well-founded orderings on sets to well-founded orderings on sequences. Given a set $S$ equipped with a well-founded ordering $>$, a status $stat$ depending on $n$ variables $x_1, \ldots, x_n$ and two lists of elements of $S$ of length $n$, we define the ordering $>_{stat}$ as follows:

$$
\{l_1, \ldots, l_n\} >_{stat} \{r_1, \ldots, r_n\}
$$

iff

$$
stat\{x_1 \mapsto l_1, \ldots, x_n \mapsto l_n\} >_{rpo} stat\{x_1 \mapsto r_1, \ldots, x_n \mapsto r_n\}
$$

As a particular case of the recursive path ordering, this ordering on sequences is of course well-founded if the starting ordering on $S$ is, and is closed under instantiation [13]. Note that this ordering boils down to the usual lexicographic or multiset ordering in case the status is a term of height one.

**General recursive schema**

**Definition 22.** Let $HOR$ be an algebraic higher-order rewrite system on $T(\mathcal{F}, \mathcal{X})$, all function symbols in $\mathcal{F}$ being equipped with a status. Let $F$ be a new functional constant, i.e. $F \not\in \mathcal{F}$, with status $stat_F$. We define $F$ with a finite set of higher-order algebraic rewrite rules satisfying the following General (recursive) Schema:

$$
F(l[X, x], Y, y) \rightarrow v
$$
with the following properties:

- \( X \subseteq Y \), and
- each subterm of \( t \) headed by \( F \) is of the form \( F(r[X,x], Y, y) \) where terms in \( r \) contain free variables only and satisfy \( I \succeq_{\text{stat}} r \).

We will term the general schema the \textit{multiset schema} (resp. \textit{lexicographic schema}) when all statuses are multiset (resp. lexicographic).

The multiset schema was first presented (with an additional superfluous hypothesis) in [32] under the name of \textit{general schema}. Note that the multiset schema is primitive recursive, because terms in \( I \) may not contain any occurrence of \( F \) by definition, hence the same is true of terms in \( r \) due to the multiset comparison. This is no more true in case of a lexicographic comparison, hence embedded recursive calls become possible for the lexicographic schema, henceforth for the general schema.

Note that there may be lambda terms in \( I, r_1, \ldots, r_m \). There are three restrictions in our schema: mutually recursive definitions are forbidden, since we define a new constant \( F \) each time; the higher-order variables in \( I \) must also appear as immediate subterms on the left-hand side definition of \( F \); terms in \( r_i \) must be smaller in a very precise sense (using the subterm relationship) than terms in \( I \). The first restriction can be easily removed, by introducing product types and packing mutually recursive definitions in a same product as done in [20] for the first-order case. The second appears to be absolutely crucial, as pointed out by our second introductory example in Section 4.6, and by its use in the proof. Most practical examples do satisfy it. The third appears to be important as well, although we believe it can be relaxed along the lines discussed in Section 4.6.

\textbf{Strong Normalization:} This section follows the same line as Section 4.3. In contrast, however, we will consider Curried \( \lambda \)-terms over the user’s signature via their encoding in the extended signature introduced in Section 3.3 and the use of Lemma 3. Hence, the rewrite relation \( \rightarrow_{\text{mix}} \) is equal to \( \rightarrow_{\beta} \cup \rightarrow_{\eta} \cup \rightarrow_{A_lq} \cup \rightarrow_{R_{lq}} \cup \rightarrow_{HOR_{lq}} \), and we will simplify the writing by dropping the superfluous \( A_lq \) subscripts. The strong normalization proof is again done by reducing the computability property to the principal case, but there will be several complications due to the use of Curried \( \lambda \)-terms on the one hand, and to the higher-order rules on the other.

\textbf{Lemma 23} (Higher-order monomorphic case). \textit{Assume \( F_1, \ldots, F_n \) are new functional constants successively added to the signature \( \mathcal{F} \), together with higher-order rules obeying the general schema. Assume also that first-order reductions are conservative. Then, for any term \( u[x_1, \ldots, x_n] \) of type \( \sigma \), whose free variables are \( x_1 : \sigma_1, \ldots, x_n : \sigma_n \), if \( \gamma = \{ x_1 : \sigma_1 \mapsto u_1, \ldots, x_n : \sigma_n \mapsto u_n \} \) is computable, then \( w = u_\gamma \) is computable.}

\textbf{Proof.} We show the result as before by a reduction to the principal case. In order to account for higher-order reductions, the proof of Lemma 4.3 must be slightly modified, by interpreting the term \( w = u_\gamma \) by the triple \( \langle M_u, u, \{ \gamma \} \rangle \), where
- To any \( F_i^j \in \mathcal{G} \) (resp. \( f^j \)) originating from some higher-order (resp. first-order) function symbol \( F_i \in \mathcal{F}_{\sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma} \), we associate the pair \((i, n-j)\) (resp. \((0, n-j)\)). 

\( M_u \) is the multiset of such pairs obtained from the multiset of function symbols occurring in \( u \). Remark that \( M_u \) contains \( M_v \) whenever \( u \gg v \), hence the arguments of Lemma 4.3 for the corresponding cases need not be repeated. Since the pairs \((i, n-j)\) order the function symbols in \( \mathcal{G} \) as the decreasing sequence \( \ldots, F_i^0, \ldots, F_i^{n-1}, F_i^n = F_{i-1}, \ldots \), the rules in \( Alg \) for \( F_i^j \) will keep \( i \) unchanged, but decrease \( n-j \).

- \( \{\gamma\} \) is the list of terms obtained from \( t \gamma \) by searching \( t \) depth first: \( \{t_1|_{p_1}, \ldots, t_j|_{p_1}\} \).

The triples are compared in the ordering 
\[ (((>N, >\gamma)_{lex})_{mul}, \gg), (\neg \text{mix} \cup \gg)_{out} \] 
\( lex \), denoted by \( \gg \) in the following and by \( \gg \) to indicate that the \( n \)-th element of the triple has decreased but not the \((n-1)\) first ones. Here, \( stat \) stands for the status \( stat_u \) of the second element \( u \) in the triple \(< M_u, u, \{\gamma\}> \). Since triples are compared lexicographically, this status is the same in both triples when the comparison reaches \( \{\gamma\} \) (as in cases 2(a) and 2(j) below), because both second elements of the triples must then be equal modulo variable renaming (this is why we keep \( u \) in the interpretation). Besides, since \( \neg \text{mix} \) is well-founded on \( \gamma \) by assumption, and because the union of the strict subterm relationship with a rewrite relation is well-founded [14], the relation \( (\neg \text{mix} \cup \gg)_{stat} \) is well-founded on \( \gamma \). Hence, \( \gg \) is a well-founded ordering on the triples.

For simplicity, we abuse the notations by comparing terms, instead of their interpretations. Compared with Lemma 13, there are five new cases in the proof, three for dealing with Curried \( \lambda \)-terms, and two for the higher-order function symbols.

1. Let \( u \) be a non-neutral term. There are two cases:
   - Let \( u \) be an abstraction. No change.
   - Let \( u \) be of the form \( F_i^j(t_1, \ldots, t_j) \) (or \( f^j(t_1, \ldots, t_j) \)). The proof is similar to the previous case. By property (C5), we need to show that \( (u\gamma t) \) is computable for all computable \( t \). By induction hypothesis, \( t_k\gamma \) is computable for all \( k \in [1..j] \). Let \( \gamma' \) be the substitution equal to \( \gamma \cup \{z \mapsto t\} \), where \( z \) is a fresh variable. By induction hypothesis again, \( F_i^{j+1}(t_1, \ldots, t_j,x)\gamma' \) is computable since \( F_i^j(t_1, \ldots, t_j) \gg F_i^{j+1}(t_1, \ldots, t_j,z)\gamma' \). We can therefore apply Lemma 12 and conclude that \( u\gamma \) is computable.

2. Let \( u \), hence \( w \) be neutral. We prove that it is computable by using property (C4), that is for every \( w \) such that \( w \longrightarrow_{\text{mix}}^\prime w' \), then \( w' \) is computable.
   - Let \( p \not\in \mathcal{F}\mathcal{P}\mathcal{S}(u) \). No change.
   - Let \( p \in \mathcal{F}\mathcal{P}\mathcal{S}(u) \), with \( p \neq A \). Although there is no change, it should be noted that any higher-order function symbol \( F_i^j \) occurring in \( u[z]_p \gamma' \) at a position of \( u[z]_p \gamma' \) occurs in \( u\gamma \) at a position of \( u \), since \( u[z]_p \gamma' = u\gamma \). This ensures that the first element of the triple does not increase from \( u\gamma \) to \( u[z]_p \gamma' \). We will not repeat this remark in the “No change” cases.
   - Let \( p = A, u = (\lambda x.v)u \) and \( w = (\lambda x.v)\gamma t \). No change.
   - Let \( p = A, u = y_t, y\gamma = \lambda x.v \). No change.
(e) Let \( p = A, u = f(u_1, \ldots, u_n) \) and \( u \) is not a variable renaming of \( v = f(x_1, \ldots, x_n) \). No change.

(f) Let \( p = A, u = f(x_1, \ldots, x_n) \). No change.

(g) Let \( p = \lambda, u = f(x_1, \ldots, x_n) \).

(h) Let \( p = \lambda, u = f(x_1, \ldots, x_n) \).

(i) Let \( p = A, u = f_{\lambda_1}(u_1, \ldots, u_n) \) and \( u \) is not a variable renaming of \( v = f_{\lambda_2}(u_1, \ldots, u_n) \). Then we proceed as in case 2(e).

(j) Let \( p = A, u = f_{\lambda_1}(x_1, \ldots, x_n) \) and \( u = f_{\lambda_2}(x_1, \ldots, x_n) \).

We first prove the following property (P): \( \sigma \) is computable for all terms \( s \) and computable substitutions \( \sigma \) such that

(i) \( F_k \) does not occur in \( s \) for \( k > i \), nor \( F_i \) for \( j < n \).

(ii) For all subterms \( F_{i \lambda}(s_1, \ldots, s_n) \) of \( s \), then \( n > 0 \) and \( \{I_{i \lambda}, \ldots, I_{n \lambda}\} \subseteq \{s_1, \ldots, s_n\} \).

We now proceed with the proof of (P) by induction on the size of \( s \). The case where \( s \) is a variable is trivial. If no \( F_{i \lambda} \) occurs in \( s \), then \( s \sigma \) is computable by the main induction hypothesis. This takes care of the case where \( s \) is a constant thanks to assumptions (i) and (ii). Otherwise, we choose an innermost subterm \( s \sigma \) and show that it is computable in order to move it to the substitution \( \sigma \), resulting in a computable substitution \( \tau \) by letting \( z, \tau = z, \sigma \), and \( \tau = (s \sigma) \) for some new variable \( z \). The pair \( (s, \tau) \) satisfies all properties required for applying (P), since so does \( (s, \tau) \). As \( s \) has a size strictly less than that of \( s \), we can apply the induction hypothesis for (P) and conclude that \( s \sigma \) is computable.

We are left to show that the innermost subterm \( F_{i \lambda}(s_1, \ldots, s_n) \) is computable. By assumption (i), the terms \( s_j \), for \( j \in [1, n] \), cannot contain any \( F_k \) for \( k > i \) or \( F_i \) for \( j < n \). By assumption that \( s \sigma \) is innermost, they cannot contain any \( F_{i \lambda} \). Hence, the terms \( s_j \sigma \) are computable by the main induction hypothesis and the substitution \( \sigma' = \{z_1 \mapsto s_1 \sigma, \ldots, z_n \mapsto s_n \sigma\} \) is computable. By assumption (ii), \( \{z_1, \ldots, z_n\} \subseteq \{I_{i \lambda}, \ldots, I_{n \lambda}\} \). Thus, \( s \sigma' \) is computable. A consequence \( F_i(z_1, \ldots, z_n) \) is computable by the main induction hypothesis.

We now show that we can apply (P) to the pair \((v, \gamma')\).
Terms in $Y \gamma', y \gamma', y y'$ are all terms in $\{y\}$, hence are computable by hypothesis. Since $X \subseteq Y$, so are terms in $X y'$. Terms in $I[X,x] y'$ being computable, they are strongly normalizable by property (C1). So, terms in $x y'$ are strongly normalizable as subterms of strongly normalizable terms. Since they are of basic type, they are computable by (C2). Hence $y'$ is computable.

Properties (i) and (ii) both come from the definition of the schema. In particular, since $\{l_1, \ldots, l_n\} \rightarrow_{\text{stat}_f} \{r_1, \ldots, r_n\}$ for any subterm $F_i(r[X,x], Y, y)$ of $v$ by hypothesis, then $\{l_1 y', \ldots, l_n y'\} \rightarrow_{\text{stat}_f} \{r_1 y', \ldots, r_n y'\}$ by closure of $\rightarrow_{\text{stat}_f}$ under instantiation.

So, the pair $(v, y')$ satisfies all requirements for (P), hence $v y'$ is computable.

Lemma 24 (Principal Case). Provided first-order algebraic reductions are conservative, any instance of a principal term $f(x_1, \ldots, x_n)$ is strongly normalizable if its alien subterms are strongly normalizable.

Proof. The previous induction for the principal case (Lemma 15) is slightly modified by considering pairs $(\text{alien}(w), \text{cap}(w))$, with the associated lexicographic ordering $((\rightarrow_{\text{mix}} \cup \rightarrow)_{\text{mul}}, \rightarrow_{\text{lex}})$. Due to the conservativity assumption of first-order algebraic reductions, rewriting in the cap does not increase the multiset of alien subterms: if $u \rightarrow_R d$ with $p \in \text{Pos}(\text{cap}(u))$, then $\text{alien}(v) \subseteq \text{alien}(u)$. As a consequence, rewriting in the cap will decrease our ordering. Now, rewriting inside an alien subterm of $w$ will clearly decrease the first argument in the pair. Note the need of using subterm in the ordering.

We can now state the main result of this paper.

Theorem 25. Let $F_1, \ldots, F_n$ be new functional constants successively added to the signature $\mathcal{F}$, together with higher-order rules obeying the general schema. Then conservative Curried (resp. algebraic) reductions are strongly normalizing provided that the set $R$ of first-order rules is terminating.

The structure of this proof is very similar to the previous one. Indeed, the computability predicate remains the same. This suggests a generalization of the result by abstracting from a particular lambda-calculus (and predicate). Work in this direction has been done by Barbanera and Fernandez [3, 2, 4]. Note also the use of the first condition of the general schema at step 2(j) of the proof to show that the substitution $X y'$ is computable. Any condition that would allow for such a conclusion could of course be used instead of (or in combination with) this requirement.

Alternative schemas. It is very attractive to extend our higher-order rewriting schema by using orderings more general than the subterm ordering, since there are many useful well-founded orderings used in proof theory and term rewriting, such as the simplifi-
cation orderings [13] and the ordinal notations. Indeed, the definition of our schema looks very much like a particular case of a higher-order rewrite rule whose termination would be proved by a recursive path ordering like ordering on higher-order terms.

There are two obstacles in this direction: reasonable orderings for higher-order terms do not exist. The only tentative we know considers β-normal, η-expanded terms [36,38], hence does not fit with our problem. Besides, in the above proof of strong normalization, the ordering needs to be compatible with the reduction ordering on alien(u). Although the subterm ordering is always compatible with reductions, it is not easy to check the compatibility for other orderings. Indeed, it is not known how to combine an ordering on terms with β-reduction.

Although we believe that a general answer exists to this problem, and we are currently working on it, we can however easily prove the result for a simple, but rather weak, modification of the schema. Assume that the termination of algebraic (first order) rewriting system is proved by the usual embedding method with a well-founded ordering >. Assume furthermore that the higher-order variables do not appear as subterms of I, and that the first-order variables in the recursive call are allowed to be substituted by algebraic terms only. Then,

Theorem 26. The strong normalization theorem holds for the above modified schema.

This schema provides, for example, a special case of transfinite recursion on the simplification orderings or the ordinal notations. (See [15,33] for the relations between these concepts.)

Conditional Schema Modulo a Congruence. So far, we have not considered conditional rules, nor rewriting modulo a congruence. There is actually no big difficulty with handling such rules, provided the general schema applies to the conditions as well on the one hand, and the congruence is compatible with the interpretation used for building our inductive argument on the other.

Definition 27. A conditional (higher-order) rewrite system R (HOR) is a set \( \{ \rho_i \} \) of conditional rules of the form

\[ \rho = \mathop{\downarrow} \frac{v_1 \wedge \cdots \wedge v_n}{\mathop{\downarrow} \mathop{\downarrow} u_1 \vdash \mathop{\downarrow} \mathop{\downarrow} v_n} \]

where \( \downarrow \) is interpreted as joinability with respect to R.

To the rule \( \rho \), we associate the term rewriting system \( \hat{\rho} = \{ l \rightarrow r, l \rightarrow u_1, l \rightarrow v_1, \ldots, l \rightarrow u_n, l \rightarrow v_n \} \), and to the set \( R = \{ \rho_i \} \) of rules we associate the set \( \hat{R} = \bigcup_i \hat{\rho}_i \) of rules.

We say that R is reductive (resp. conservative) if \( \hat{R} \) is terminating (resp. conservative). Reductive systems are of course terminating [34].

We say that HOR satisfies the general schema iff the associated rewrite system HÒR satisfies the general schema.

Our result generalizes directly to this case.
Associative–commutative rewriting is very important in practice. We can also generalize our results to associative–commutative rewriting, even if there are higher-order associative–commutative function symbols. Of course, a decidable pattern matching algorithm will be necessary, and we know that it is the case for second-order definitions [12]. More generally, we can generalize our strong normalization result to rewriting modulo a congruence $E$ whose equivalence classes have adequate properties. It is important here that the subterm and encompassment orderings are still compatible (with respect to well-foundedness) with the rewriting relation on $E$-congruence classes of terms, which in turn requires that the congruence classes are finite [30]. This implies in particular that this congruence is generated by regular equations, that is equations $l = r$ such that $\text{Var}(l) = \text{Var}(r)$. It is also important that equivalent terms are built up from the same set of function symbols in order for our interpretation to be compatible with the congruence, and for the principal lemma to work. This does not imply any particular restriction for the first-order function symbols, but it does for the higher-order ones: besides the regularity condition, the axioms in $E$ should be built up with function symbols belonging to the same level of the hierarchy. This is the case with associativity and commutativity, and more generally with permutative axioms.

Our results again generalize directly to this case, and actually to the combination of conditional rewriting and rewriting modulo satisfying the above restrictions.

5. Confluence of abstract data type systems

We can now turn our attention to the Church–Rosser property, in case of higher-order rules satisfying the general schema. Since mixed conservative reductions are strongly normalizing, we can use Newman's lemma and analyze the Church–Rosser property in terms of critical pairs. Note that the existence of new critical pairs depends solely on left-hand sides of rules. As a consequence, whether the higher-order rules follow the multiset or lexicographic schema does not matter in this respect.

By introducing the application operator explicitly, the Church–Rosser property can be easily reduced to the first-order case. This shows in particular that higher-order critical pairs need to consider overlaps on higher-order variables.

We will of course have two different kinds of confluence results, one for algebraic $\lambda$-terms for which $\eta$-reductions are allowed, and another one for mixed $\lambda$-terms for which $\eta$-reductions are not allowed.

**Proposition 28.** Let $F_1, \ldots, F_n$ be new functional constants successively added to the signature $\mathcal{F}$ together with higher-order rules obeying the general schema. Then conservative Curried reductions (excluding $\eta$) are confluent on Curried $\lambda$-terms, provided that the set $R$ of first-order rules is confluent, and there are no critical pairs between the higher-order rules, and between the first-order rules and the higher-order rules.
Proposition 29. Let $F_1, \ldots, F_n$ be new functional constants successively added to the signature $\mathcal{F}$ together with higher-order rules obeying the general schema. Then conservative algebraic reductions (including $\eta$) are confluent on algebraic $\lambda$-terms, provided that the set $R$ of first-order rules is confluent, and there are no critical pairs between the higher-order rules, and between the first-order rules and the higher-order rules.

In both cases, the proof is a routine inspection of all divergent reductions with algebraic rules. Since these reductions do not form critical pairs, the result follows. For example, Gallier’s counterexample to confluence of the rewrite system $f(x) \rightarrow a$ in presence of $\eta$:

$$f \rightarrow_\eta \lambda x. (fx) \rightarrow_R \lambda x. a$$

does not apply anymore when considering algebraic $\lambda$-terms, because the mixed term $(fx)$ does not exist anymore. Indeed, if the $\eta$ rule applies to $(\lambda x. M)x$, then no rewrite rule may apply to $M$ and $x$ at the same time, hence a critical pair is not possible.

Note that there is an alternative way to achieve this goal: by using $\eta$ as an expansion (see [17]). Both approaches share the (implicit or explicit) use of $\eta$ as an expansion for algebraic terms, but our approach uses $\eta$ as a contraction for the other terms.

We now turn our attention to a subclass of the above schema, called weak schema, which is interesting for practical purposes and for which the critical pairs with the first-order rules are easily computed.

Definition 30. For each higher-order rule $F(l, Y, y) \rightarrow v$ following the general schema, the weak schema assumes the additional restriction $l \in \mathcal{F}(\mathcal{F}_1, \mathcal{A}_1)$.

On the left-hand side of the weak schema, the only occurrence of a higher-order functional constant is at the root, and higher-order variables are at the leaves. As a consequence, the only possible (non-variable) overlaps in the combined system are either overlaps between two rules defining the functional constant $F$, or overlaps of a first-order rule inside $l$. A simple example of the weak schema is the introduction of functional constants by primitive recursion (structured recursion) of higher types on a first-order data structure. In this schema, there is one higher-order rule for each constructor $c_i$ (including the constants):

$$F(c_i(x_1, \ldots, x_p), Y, y) \rightarrow v[F(x_1, Y, y), \ldots, F(x_p, Y, y), x_1, \ldots, x_p, Y, y]$$

The above primitive recursion schema has no overlap between the higher-order rules, or between the higher-order rules and the first-order ones. As a consequence, we have the corollary.

Corollary 31. Algebraic functional languages are Church–Rosser and strongly normalizing when the higher-order rules obey the structured recursion discipline.
Note that we can simply extend our weak schema by recursing on several arguments of $F$ rather than a single one as above. Our definition of the general schema clearly allows it.

6. Conclusion

The notion of abstract data type system appears to be an appealing powerful combination of the notions of abstract data type and type system in order to write specifications. The setting of abstract data types provides with what abstract data types are good for, a notion of module (or object in the OBJ jargon) allowing to structure the specification around basic types. The type system allows to express sophisticated logical properties very concisely. Moreover, both settings come along with a notion of computation by rewriting which can be easily combined while keeping the most important meta-theoretic properties of both settings. We have therefore outlined a theory of abstract data type systems that has many applications, to both type theory and abstract data types. Let us give a few example to illustrate its expressivity and weaknesses.

6.1. Algebraic functional languages

The kind of algebraic functional language investigated in Section 4.6 with first-order polymorphism and inheritance for all types is a very good candidate for a real functional programming language allowing to write higher-order pattern-matching definitions. We give below a simple example of our rewrite rule using a non-constructor term, hence it is not a structural recursion.

\[
\begin{align*}
  x + s(y) &\rightarrow s(x + y) & x + 0 &\rightarrow x \\
  \text{COMP}(X,Y) &\rightarrow \lambda z.X(Y(z)) & F(X,s(x)) &\rightarrow X(F(X,x)) \\
  F(X,0) &\rightarrow \lambda x.x & F(X,x+y) &\rightarrow \text{COMP}(F(X,x),F(X,y))
\end{align*}
\]

All higher-order rewrite rules above follow our general schema, hence the combined system of the first-order rules, the higher-order rules and the $\lambda$-calculus is strongly normalizing. In particular, the last rule uses a non-constructor term, and indeed the system is not Church–Rosser: $F(X,s(x)+s(y))$ has two normal forms $X(F(X,s(x)+y))$ and $\lambda z.X(F(x))(X(F(y)))$. Note however that it is Church–Rosser on ground terms. This does not follow from our results, of course, since critical pairs allow investigating the confluence property for open terms only.

6.2. Higher-order abstract data types

Abstract data types use normally first-order equations. This point is stressed by Goguen in many papers as an advantage. One reason is that there is an important body of work for first-order rewriting. Our results allow to free ourselves from this restriction. As a result, one can give elegant specifications of complex operations. In
particular, the use of higher-order rules allows to reuse code that would be duplicated otherwise, as illustrated by the following OBJ-like specification of sorting:

**Example 3.** The following is a rewrite program to sort a list of natural numbers by inserting elements one by one into position. cons and nil are the list constructors, and s and 0 are the constructors for natural numbers. We use x and y for arbitrary natural numbers, and l for an arbitrary list of natural numbers:

\[
\begin{align*}
\text{max}(0, x) & \rightarrow \text{x} & \text{min}(0, x) & \rightarrow 0 \\
\text{max}(x, 0) & \rightarrow \text{x} & \text{min}(x, 0) & \rightarrow 0 \\
\text{max}(s(x), s(y)) & \rightarrow s(\text{max}(x, y)) & \text{min}(s(x), s(y)) & \rightarrow s(\text{min}(x, y)) \\
\text{insert}(x, \text{nil}, X, Y) & \rightarrow \text{cons}(x, \text{nil}) \\
\text{insert}(x, \text{cons}(y, l), X, Y) & \rightarrow \text{cons}(X(x, y), \text{insert}(Y(x, y), l, X, Y)) \\
\text{sort}(\text{nil}, X, Y) & \rightarrow \text{nil} \\
\text{sort}(\text{cons}(x, l), X, Y) & \rightarrow \text{insert}(x, \text{sort}(l, X, Y), X, Y) \\
\text{ascending_sort}(l) & \rightarrow \text{sort}(l, \lambda x.\lambda y.\text{min}(x, y), \lambda x.\lambda y.\text{max}(x, y)) \\
\text{descending_sort}(l) & \rightarrow \text{sort}(l, \lambda x.\lambda y.\text{max}(x, y), \lambda x.\lambda y.\text{min}(x, y))
\end{align*}
\]

Here, insert must have a lexicographic status, from right to left, while min, max and sort may have an arbitrary status. Note our use of \(\eta\) expansions \(\lambda x.\lambda y.\text{min}(x, y)\) to stick with algebraic \(\lambda\)-terms.

### 6.3. Inductive types

As explained in the introduction, inductive types give rise to higher-order rewrite rules for the associated recursor. We already gave the example of the natural numbers. Let us now first consider a recursor (of a certain type) for the inductive type of lists defined by the constructors nil and cons:

\[
\text{Rec}(X, Y, x, \text{cons}(y, l)) \rightarrow Y(y, l, \text{Rec}(X, Y, x, l)) \quad \text{Rec}(X, Y, x, \text{nil}) \rightarrow X(x)
\]

As easily seen, this type of introduction of the recursors follows our weak schema introduced in Section 5, hence preserves strong normalization and confluence. This is not true, however, in general, as exemplified by a more complex example taken from [47], a specification of “ordinal numbers” (this is not quite true, of course), for which there are three constructors, 0, succ, and lim which constructs the “ordinal” which is the limit of an denumerable sequence of “ordinals”:

OBJ Ord

\[
\text{op 0} : \text{Ord}
\]
op succ : Ord → Ord
op lim : (Nat → Ord) → Ord
end OBJ

Note here that the first argument of the constructor lim has a functional type. The so-called transfinite induction axiom for this type is the following:

\[
\text{Ind} \overset{\text{def}}{=} \forall P. P(0) \rightarrow \forall x. [P(x) \rightarrow P(\text{succ}(x))] \\
\rightarrow \forall f. [\forall n. P(f(n))] \rightarrow P(\text{lim } f) \rightarrow \forall y. P(y)
\]

where \(n\) has type Nat and \(f\) has type Nat → Ord.

Now, we proceed with the construction of the rules for the recursor rec:

\[
\text{rec } P \; t \; u \; v \; 0 \rightarrow t \\
\text{rec } P \; t \; u \; v \; \text{succ}(x) \rightarrow (u \; x \; (\text{rec } P \; t \; u \; v \; x)) \\
\text{rec } P \; t \; u \; v \; (\text{lim } f) \rightarrow (v \; f \; \lambda n. (\text{rec } P \; t \; u \; v \; (f \; n)))
\]

The right-hand side of the third rule includes an abstraction. This is not forbidden by our general schema, but the abstraction takes place outside the recursive call and binds a variable inside the recursive call, which is explicitly forbidden. As a consequence, there is no way to say that the recursive call is smaller than the left-hand side call in our ordering, nor in any syntactic ordering. However, it is clearly smaller in some sense, since \((\text{lim } f)\) is the "ordinal" limit of the sequence \((f \; n)\), hence is the collection of all the \((f \; n)\). By saying this however, we implicitly refer to the standard model of the abstract data type Ord. This suggests to use interpretation orderings, rather than syntactic orderings to define a more elaborated schema. This approach is taken by Werner to show the strong normalization theorem for the calculus of inductive constructions [47].

On the other hand, inductive types whose all constructors have a type of the form \(\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma\), such that \(\sigma_1, \ldots, \sigma_n\) are all basic types, follow the general schema. A typical example is the type of natural numbers which we gave in the introduction.

Note also that there are several possibilities for defining the recursor, corresponding to different ways to eliminate the corresponding cuts. We did not investigate yet whether there is a way for defining the recursor that would satisfy our general schema, or at least be easier to work with. This is left for future work.

6.4. Concluding remarks

The theory of combining rule-based definitions with \(\lambda\)-calculus has made enough progress so as to start implementing new languages encompassing both words. Finally, note that we have not fulfilled all promises, since we do not use the full power of equations when computations are restricted to rewriting: equations allow logical variables, by performing narrowing instead of rewriting. Our last guess is that extending this framework to narrowing should provide the adequate framework for mixing equality
and functions. This would be a key step in providing a unified framework for hosting algebraic, functional and logic programming styles.

Acknowledgements

To Maribel Fernandez and Ralph Matthes for scrutinizing several versions of this work.

References


