Inductive types in the Calculus of Algebraic Constructions

Frédéric Blanqui

Fundamenta Informaticae 65(1-2), 61-86, 2005
IOS Press

Inductive types in the Calculus of Algebraic Constructions

Frédéric Blanqui
LORIA & INRIA
615 rue du Jardin Botanique, BP 101, 54602 Villers-lès-Nancy, France
http://www.loria.fr/~blanqui/ - blanqui@loria.fr

Abstract. In a previous work, we proved that an important part of the Calculus of Inductive Constructions (CIC), the basis of the Coq proof assistant, can be seen as a Calculus of Algebraic Constructions (CAC), an extension of the Calculus of Constructions with functions and predicates defined by higher-order rewrite rules. In this paper, we prove that almost all CIC can be seen as a CAC, and that it can be further extended with non-strictly positive types and inductive-recursive types together with non-free constructors and pattern-matching on defined symbols.

1. Introduction

There has been different proposals for defining inductive types[1] and functions in typed systems. In Girard's polymorphic λ-calculus or in the Calculus of Constructions (CC) [10], data types and functions can be formalized by using impredicative encodings, difficult to use in practice, and computations are done by β-reduction only. In Martin-Löf's type theory or in the Calculus of Inductive Constructions (CIC) [11], inductive types and their induction principles are first-class objects, functions can be defined by induction and computations are done by ι-reduction, the rules for cut-elimination in inductive proofs. For instance, for the type nat of natural numbers, the recursor[2] $rec : (P : nat \Rightarrow \star)(u : P0)(v : (n : nat)\ Pn \Rightarrow P(sn))(n : nat)Pn$ is defined by the following ι-rules:

$$\begin{aligned} rec\ P\ u\ v\ 0 &\ \to_\iota\ u \\ rec\ P\ u\ v\ (s\ n) &\ \to_\iota\ v\ n\ (rec\ P\ u\ v\ n) \end{aligned}$$

Finally, in the algebraic setting [12], functions are defined by using rewrite rules and computations are done by applying these rules. Since both β-reduction and ι-reduction are particular cases of higher-order rewriting [18], proposals soon appeared for integrating all these approaches. Starting with [16, 2],

[1] All over the paper, by "inductive types", we also mean inductively defined predicates or families of types.
[2] $(x : T)P$ is a usual type-theoretic notation for the dependent product or universal quantification "for all x of type T, P".

this objective culminated with [4, 5, 6] in which an important part of CIC (described in [5]) can be seen as a Calculus of Algebraic Constructions (CAC), an extension of CC with functions and predicates defined by higher-order rewrite rules. In this paper, we go one step further in this direction, capture almost all CIC and extend it with non-strictly positive inductive types and inductive recursive types [13].

Let us see two examples of recursors that are allowed in CIC but not in CAC [26]. The first example is a third-order definition of finite sets of natural numbers (represented as predicates over nat):

$$\begin{aligned}
\mathit{fin} &: (nat \Rightarrow \star) \Rightarrow \star \\
\mathit{empty} &: \mathit{fin}([y : nat]\bot) \\
\mathit{add} &: (x : nat)(p : nat \Rightarrow \star)\mathit{fin}\, p \Rightarrow \mathit{fin}([y : nat]y = x \vee (p\ y)) \\
\mathit{rec} &: (Q : (nat \Rightarrow \star) \Rightarrow \star)Q([y : nat]\bot) \\
&\quad \Rightarrow ((x : nat)(p : nat \Rightarrow \star)\mathit{fin}\, p \Rightarrow Qp \Rightarrow Q([y : nat]y = x \vee (p\ y))) \\
&\quad \Rightarrow (p : nat \Rightarrow \star)\mathit{fin}\, p \Rightarrow Qp
\end{aligned}$$

where $\bot$ is the false proposition and the *weak* recursor rec, *i.e.* the recursor for defining objects, is defined by the rules:

$$\begin{aligned}
rec\, Q\ u\ v\ p'\ \mathit{empty} &\ \rightarrow\ u \\
rec\, Q\ u\ v\ p'\ (\mathit{add}\ x\ p\ h) &\ \rightarrow\ v\ x\ p\ h\ (rec\, Q\ u\ v\ p\ h)
\end{aligned}$$

The problem comes from the fact that, in the output type of add, $\mathit{fin}([y : nat]y = x \vee (p\ y))$, the predicate p is not parameter of fin. This is why the corresponding *strong* recursor, *i.e.* the recursor for defining types or predicates, is not allowed in CIC (p could be "bigger" than fin) [9]. This can be generalized to any big/impredicative dependent type, that is, to any type having a constructor with a predicate argument which is not a parameter. Formally, this condition, called **(I6)** in [6], *safeness* in [29] and *$\star$-dependency for constructors* in [31], can be stated as follows:

Definition 1.1. (I6)
If $C : (\vec{z} : \vec{V})\star$ is a type and $c : (\vec{x} : \vec{T})C\vec{v}$ is a constructor of C then, for all predicate variable x occurring in some T_j, there is some argument $v_{\iota_x} = x$.

The second example is John Major's equality which is intended to equal terms of different types [20]:

$$\begin{aligned}
\mathit{JMeq} &: (A : \star)A \Rightarrow (B : \star)B \Rightarrow \star \\
\mathit{refl} &: (C : \star)(x : C)(\mathit{JMeq}\ C\ x\ C\ x) \\
\mathit{rec} &: (A : \star)(x : A)(P : (B : \star)B \Rightarrow \star)(P\ A\ x) \\
&\quad \Rightarrow (B : \star)(y : B)(\mathit{JMeq}\ A\ x\ B\ y) \Rightarrow (P\ B\ y)
\end{aligned}$$

where rec is defined by the rule:

$$rec\, C\ x\ P\ h\ C\ x\ (\mathit{refl}\ C\ x)\ \rightarrow\ h$$

Here, the problem comes from the fact that, in the output type of $refl$, the argument for B is equal to the argument for A. This can be generalized to any polymorphic type having a constructor with two equal type parameters. From a rewriting point of view, this is like having pattern-matching or non-linearities on predicate arguments, which is known to create inconsistencies in some cases [15]. A similar restriction called *$\star$-dependency for function symbols* also appears in [31].

Definition 1.2. (Safeness)
A rule $f\vec{l} \to r$ with $f : (\vec{x} : \vec{T})U$ is *safe* if:

- for all predicate argument x_i, l_i is a variable,
- if x_i and x_j are two distinct predicate arguments, then $l_i \neq l_j$.

An inductive type is *safe* if the corresponding ι-rules are safe.

By using what is called in Matthes' terminology [19] an *elimination-based* interpretation instead of the *introduction-based* interpretation that we used in [6], we prove that weak recursors for types like fin or $JMeq$ can be accepted, hence that CAC subsumes CIC almost completely. The only condition we could not get rid of is the safeness condition for predicate-level rewrite rules. So, we do not accept strong elimination on $JMeq$ (strong elimination for fin is allowed neither in CIC nor in CAC [9]). On the other hand, we prove that CAC and CIC can be easily extended to non-strictly positive types (Section 8) and to inductive-recursive types (Section 9) [13].

2. The Calculus of Inductive Constructions (CIC)

We assume the reader familiar with typed λ-calculi [3]. In this section, we present CIC as defined in [32]. In order to type the strong elimination schema in a polymorphic way, which is not possible in CC, Werner uses a slightly more general Pure Type System (PTS) [3]. CC is the PTS with the sorts $\mathcal{S} = \{\star, \Box\}$, the axioms $\mathcal{A} = \{(\star, \Box)\}$ and the rules $\mathcal{B} = \{(s_1, s_2, s_3) \in \mathcal{S}^3 \mid s_2 = s_3\}$. Werner extends it by adding the sort $\triangle$, the axiom $(\Box, \triangle)$ and the rules $(\star, \triangle, \triangle)$ and $(\Box, \triangle, \triangle)$. In fact, he denotes $\star$ by *Set*, $\Box$ by *Type* and $\triangle$ by *Extern*. The sort $\star$ denotes the universe of types and propositions, and the sort $\Box$ denotes the universe of predicate types (also called *kinds*). For instance, the type nat of natural numbers is of type $\star$, $\star$ itself is of type $\Box$ and $nat \Rightarrow \star$, the type of predicates over nat, is of type $\Box$. Then, Werner adds terms for representing inductive types, their constructors and the definitions by recursion on these types:

- **Inductive types.** An inductive type is denoted by $I = Ind(X : A)\{\vec{C}\}$ where $\vec{C}$ is an ordered sequence of terms for the types of the constructors of I. For instance, $Nat = Ind(X : \star)\{X, X \Rightarrow X\}$ represents the type of natural numbers (in fact, any type isomorphic to the type of natural numbers). The term A must be of the form $(\vec{x} : \vec{A})\star$ and the C_i's of the form $(\vec{z} : \vec{B})X\vec{m}$ with no X in $\vec{m}$. Furthermore, the inductive types must be strictly positive. In CIC, this means that, if $C_i = (\vec{z} : \vec{B})X\vec{m}$ then, for all j, either X does not occur in B_j, or B_j is of the form $(\vec{y} : \vec{D})X\vec{q}$ and X occurs neither in $\vec{D}$ nor in $\vec{q}$.
- **Constructors.** The i-th constructor of an inductive type I is denoted by $Constr(i, I)$. For instance, $Constr(1, Nat)$ represents zero and $Constr(2, Nat)$ represents the successor function.
- **Definitions by recursion.** A definition by recursion on an inductive type I is denoted by $Elim(I, Q, \vec{a}, c)$ where Q is the type of the result, $\vec{a}$ the arguments of I and c a term of type $I\vec{a}$. The strong elimination (*i.e.* when Q is a predicate type) is restricted to *small* inductive types, that is, to the types whose

constructors have no other predicate arguments than the ones that their type have. Formally, an inductive type $I = Ind(X : A)\{\vec{C}\}$ is *small* if all the types of its constructors are small, and a constructor type $C = (\vec{z} : \vec{B})X\vec{m}$ is *small* if $\vec{z}$ are object variables (this means that the predicate arguments must be part of the environment in which they are typed; they cannot be part of $\vec{C}$).

For defining the reduction relation associated with $Elim$, called *ι-reduction* and denoted by $\to_\iota$, and the typing rules of these inductive constructions (see Figure 1), it is necessary to introduce a few definitions. Let C be a constructor type. We define $\Delta\{I, X, C, Q, c\}$ as follows:

- $\Delta\{I, X, X\vec{m}, Q, c\} = Q\vec{m}c$
- $\Delta\{I, X, (z : B)D, Q, c\} = (z : B)\Delta\{I, X, D, Q, cz\}$ if X does not occur in B
- $\Delta\{I, X, (z : B)D, Q, c\} = (z : B\{X \mapsto I\})((\vec{y} : \vec{D})Q\vec{q}(z\vec{y})) \Rightarrow \Delta\{I, X, D, Q, cz\}$ if $B = (\vec{y} : \vec{D})X\vec{q}$

Then, the *ι-reduction* is defined by the rule:

$$Elim(I, Q, \vec{x}, Constr(i, I')\vec{z})\{\vec{f}\} \ \to_\iota \ \Delta[I, X, C_i, f_i, FunElim(I, Q, \vec{f})]\vec{z}$$

where $I = Ind(X : A)\{\vec{C}\}$, $FunElim(I, Q, \vec{f}) = [\vec{x} : \vec{A}][y : I\vec{x}]Elim(I, Q, \vec{x}, y)\{\vec{f}\}$ and $\Delta[I, X, C, f, F]$ is defined as follows:

- $\Delta[I, X, X\vec{m}, f, F] = f$
- $\Delta[I, X, (z : B)D, f, F] = [z : B]\Delta[I, X, D, fz, F]$ if X does not occur in B
- $\Delta[I, X, (z : B)D, f, F] = [z : B\{X \mapsto I\}]\Delta[I, X, D, fz[\vec{y} : \vec{D}](F\vec{q}(z\vec{y})), F]$ if $B = (\vec{y} : \vec{D})X\vec{q}$

Finally, in the type conversion rule (Conv), in addition to β-reduction and ι-reduction, Werner considers η-reduction: $[x : T]ux \to_\eta u$ if x does not occur in u. The relation $\leftrightarrow^*_{\beta\eta\iota}$ is the reflexive, symmetric and transitive closure of $\to_{\beta\eta\iota}$. Note that, since $\to_{\beta\eta}$ is not confluent on badly typed terms [23], considering η-reduction creates important difficulties.

3. The Calculus of Algebraic Constructions (CAC)

We assume the reader familiar with rewriting [12]. The Calculus of Algebraic Constructions (CAC) [6] simply extends CC with a set $\mathcal{F}$ of *symbols* and a set $\mathcal{R}$ of *rewrite rules* (see Definition 3.3).

Definition 3.1. (Terms)
The set $\mathcal{T}$ of CAC terms is inductively defined as follows:

$$t, u \in \mathcal{T} ::= s \mid x \mid f \mid [x : t]u \mid tu \mid (x : t)u$$

where $s \in \mathcal{S} = \{\star, \Box\}$ is a *sort*, $x \in \mathcal{X}$ is a *variable*, $f \in \mathcal{F}$ is a *symbol*, $[x : t]u$ is an *abstraction*, tu is an *application*, and $(x : t)u$ is a *dependent product*, written $t \Rightarrow u$ if x does not freely occur in u. As usual, terms are considered up to α-conversion, *i.e.* up to sort-preserving renaming of bound variables. A term t is *of the form* a term u if t is α-convertible to $u\sigma$ for some substitution σ.

We denote by $\mathrm{FV}(t)$ the set of variables that freely occur in t, by $\mathrm{Pos}(t)$ the set of Dewey's positions in t (words on strictly positive integers), by $t|_p$ the subterm of t at position p, by $\mathrm{Pos}(x, t)$ the set of

Figure 1. Typing rules for inductive constructions in CIC

$$\text{(Ind)}\quad \frac{A = (\vec{x} : \vec{A}) \star \quad \Gamma \vdash A : \Box \quad \forall i, \Gamma, X : A \vdash C_i : \star \quad I = Ind(X : A)\{\vec{C}\} \text{ is strictly positive}}{\Gamma \vdash I : A}$$

$$\text{(Constr)}\quad \frac{I = Ind(X : A)\{\vec{C}\} \quad \Gamma \vdash I : T}{\Gamma \vdash Constr(i, I) : C_i\{X \mapsto I\}}$$

$$\text{(}\star\text{-Elim)}\quad \frac{\begin{array}{c} A = (\vec{x} : \vec{A}) \star \quad I = Ind(X : A)\{\vec{C}\} \quad \Gamma \vdash Q : (\vec{x} : \vec{A})I\vec{x} \Rightarrow \star \\ T_i = \Delta\{I, X, C_i, Q, Constr(i, I)\} \\ \forall j, \Gamma \vdash a_j : A_j\{\vec{x} \mapsto \vec{a}\} \quad \Gamma \vdash c : I\vec{a} \quad \forall i, \Gamma \vdash f_i : T_i \end{array}}{\Gamma \vdash Elim(I, Q, \vec{a}, c)\{\vec{f}\} : Q\vec{a}c}$$

$$\text{(}\Box\text{-Elim)}\quad \frac{\begin{array}{c} A = (\vec{x} : \vec{A}) \star \quad I = Ind(X : A)\{\vec{C}\} \text{ is small} \quad \Gamma \vdash Q : (\vec{x} : \vec{A})I\vec{x} \Rightarrow \Box \\ T_i = \Delta\{I, X, C_i, Q, Constr(i, I)\} \\ \forall j, \Gamma \vdash a_j : A_j\{\vec{x} \mapsto \vec{a}\} \quad \Gamma \vdash c : I\vec{a} \quad \forall i, \Gamma \vdash f_i : T_i \end{array}}{\Gamma \vdash Elim(I, Q, \vec{a}, c)\{\vec{f}\} : Q\vec{a}c}$$

$$\text{(Conv)}\quad \frac{\Gamma \vdash t : T \quad T \leftrightarrow^*_{\beta\eta\iota} T' \quad \Gamma \vdash T' : s}{\Gamma \vdash t : T'}$$

positions $p \in \mathrm{Pos}(t)$ such that $t|_p$ is a free occurrence of x in t, and by $\mathrm{dom}(\theta) = \{x \in \mathcal{X} \mid x\theta \neq x\}$ the *domain* of a substitution θ. Let $\vec{t}$ denote a sequence of terms $t_1 \dots t_n$ of length $|\vec{t}| = n \geq 0$.

Every $x \in \mathcal{X} \cup \mathcal{F}$ is equipped with a sort s_x. We denote by $\mathcal{X}^s$ (resp. $\mathcal{F}^s$) the set of variables (resp. symbols) of sort s. Let $\mathrm{FV}^s(t) = \mathrm{FV}(t) \cap \mathcal{X}^s$ and $\mathrm{dom}^s(\theta) = \mathrm{dom}(\theta) \cap \mathcal{X}^s$. A variable or a symbol of sort $\star$ (resp. $\Box$) is an *object* (resp. a *predicate*).

Although terms and types are mixed in Definition 3.1, we can distinguish the following three disjoint sub-classes where $t \in \mathcal{T}$ denotes any term:

- objects: $o \in \mathcal{O} ::= x \in \mathcal{X}^\star \mid f \in \mathcal{F}^\star \mid [x : t]o \mid ot$
- predicates: $P \in \mathcal{P} ::= x \in \mathcal{X}^\Box \mid f \in \mathcal{F}^\Box \mid [x : t]P \mid Pt \mid (x : t)P$
- predicate types or kinds: $K \in \mathcal{K} ::= \star \mid (x : t)K$

Definition 3.2. (Precedence)
We assume given a total quasi-ordering $\geq$ on symbols whose strict part $>= \geq \setminus \leq$ is well-founded, and let $\simeq = \geq \cap \leq$ be its associated equivalence relation. A symbol f is *smaller* (resp. *strictly smaller*) than a symbol g iff $f \leq g$ (resp. $f < g$). A symbol f is *equivalent* to a symbol g iff $f \simeq g$.

Figure 2. Typing rules of CAC

$$\text{(ax)} \qquad \vdash \star : \Box$$

$$\text{(symb)} \qquad \frac{\vdash \tau_f : s_f}{\vdash f : \tau_f}$$

$$\text{(var)} \qquad \frac{\Gamma \vdash T : s_x}{\Gamma, x : T \vdash x : T} \qquad (x \notin \mathrm{dom}(\Gamma))$$

$$\text{(weak)} \qquad \frac{\Gamma \vdash t : T \quad \Gamma \vdash U : s_x}{\Gamma, x : U \vdash t : T} \qquad (x \notin \mathrm{dom}(\Gamma))$$

$$\text{(prod)} \qquad \frac{\Gamma \vdash U : s \quad \Gamma, x : U \vdash V : s'}{\Gamma \vdash (x : U)V : s'}$$

$$\text{(abs)} \qquad \frac{\Gamma, x : U \vdash v : V \quad \Gamma \vdash (x : U)V : s}{\Gamma \vdash [x : U]v : (x : U)V}$$

$$\text{(app)} \qquad \frac{\Gamma \vdash t : (x : U)V \quad \Gamma \vdash u : U}{\Gamma \vdash tu : V\{x \mapsto u\}}$$

$$\text{(conv)} \qquad \frac{\Gamma \vdash t : T \quad \Gamma \vdash T' : s}{\Gamma \vdash t : T'} \qquad (T \downarrow_{\beta\mathcal{R}} T')$$

Definition 3.3. (Rewrite rule)
The terms only built from variables and applications of the form $f\vec{t}$ are called *algebraic*. A *rewrite rule* is a pair $l \to r$ such that:

- l is algebraic,
- l is not a variable,
- $\mathrm{FV}(r) \subseteq \mathrm{FV}(l)$,
- every symbol occurring in r is smaller than f.

The rewrite relation $\to_{\mathcal{R}}$ induced by $\mathcal{R}$ is the smallest relation containing $\mathcal{R}$ and stable by context and substitution: $t \to_{\mathcal{R}} t'$ iff there exist $p \in \mathrm{Pos}(t)$, $l \to r \in \mathcal{R}$ and σ such that $t = t[l\sigma]_p$ and $t' = t[r\sigma]_p$. A symbol f with no rule $f\vec{l} \to r \in \mathcal{R}$ is *constant*, otherwise it is (partially) *defined*. Let $\mathcal{CF}^s$ (resp. $\mathcal{DF}^s$) be the set of constant (resp. defined) symbols of sort s.

Definition 3.4. (Typing)
Every $f \in \mathcal{F}$ is equipped with a *type* τ_f such that:

- τ_f is a closed term of the form $(\vec{x} : \vec{T})U$ with U distinct from a product,

– every symbol occurring in τ_f is strictly smaller than f,
– for every rule $f\vec{l} \to r \in \mathcal{R}$, we have $|\vec{l}| \leq |\vec{x}|$.

A *constructor* is any symbol f whose type is of the form $(\vec{y} : \vec{U})C\vec{v}$ with $C \in \mathcal{CF}^{\Box}$. Let $\mathcal{C}ons$ be the set of constructors. A typing *environment* is a sequence of variable-type pairs. Given f of type $(\vec{x} : \vec{T})U$, we denote by Γ_f the environment $\vec{x} : \vec{T}$.

The typing relation of CAC is the relation $\vdash$ defined in Figure 2. Let $\vdash_g$ (resp. $\vdash_g^<$) be the typing relation defined by the rules of Figure 2 with the side condition $f \leq g$ (resp. $f < g$) in the (symb) rule.

In comparison with CC, we added the rule (symb) for typing symbols and, in the rule (conv), we replaced $\downarrow_\beta$ by $\downarrow_{\beta\mathcal{R}}$, where $u \downarrow_{\beta\mathcal{R}} v$ iff there exists a term w such that $u \to^*_{\beta\mathcal{R}} w$ and $v \to^*_{\beta\mathcal{R}} w$, $\to^*_{\beta\mathcal{R}}$ being the reflexive and transitive closure of $\to_{\beta\mathcal{R}} = \to_\beta \cup \to_{\mathcal{R}}$. This means that types having a common reduct are identified and share the same proofs: any term of type T is also of type T' if T and T' have a common reduct. For instance, a proof of $P(2+2)$ is also a proof of $P(4)$ if $\mathcal{R}$ contains the rules:

$$\begin{aligned} x + 0 &\to x \\ x + (s\ y) &\to s\ (x + y) \end{aligned}$$

This decreases the size of proofs by an important factor, and increases the automation as well. **All over the paper, we assume that $\to = \to_{\beta\mathcal{R}}$ is confluent**. This is the case if, for instance, $\mathcal{R}$ is left-linear and confluent [22], like ι-reduction is.

A substitution θ *preserves typing from* Γ *to* Δ, written $\theta : \Gamma \rightsquigarrow \Delta$, if, for all $x \in \mathrm{dom}(\Gamma)$, $\Delta \vdash x\theta : x\Gamma\theta$, where $x\Gamma$ is the type associated to x in Γ. Type-preserving substitutions enjoy the following important property: if $\Gamma \vdash t : T$ and $\theta : \Gamma \rightsquigarrow \Delta$ then $\Delta \vdash t\theta : T\theta$ (Lemma 24 in [5]).

For ensuring the *subject reduction* property (preservation of typing under reduction, see Theorems 5 and 16 in [6]), rules must satisfy the following conditions (see Definition 3 in [6]):

Definition 3.5. (Well-typed rules)
Every rule $f\vec{l} \to r$ is assumed to be equipped with an environment Γ and a substitution ρ such that, if $\tau_f = (\vec{x} : \vec{T})U$ and $\gamma = \{\vec{x} \mapsto \vec{l}\}$, the following conditions are satisfied:
– $\Gamma \vdash r : U\gamma\rho$,
– $\forall \Delta, \sigma, T$, if $\Delta \vdash l\sigma : T$ then $\sigma : \Gamma \rightsquigarrow \Delta$ and $\sigma \downarrow \rho\sigma$.

The first condition is decidable under the quite natural restriction that the typing of r does not need the use of $f\vec{l} \to r$. The other conditions generally follow from the inversion of the judgment $\Delta \vdash l\sigma : T$, and confluence for the condition $\sigma \downarrow \rho\sigma$. Lemma 7 in [6] gives sufficient conditions for deciding that $\sigma : \Gamma \rightsquigarrow \Delta$.

The substitution ρ allows to eliminate non-linearities only due to typing. This makes rewriting more efficient and the proof of confluence easier. For instance, the concatenation on polymorphic lists (type $list : \star \Rightarrow \star$ with constructors $nil : (A : \star)listA$ and $cons : (A : \star)A \Rightarrow listA \Rightarrow listA$) of type $(A : \star)listA \Rightarrow listA \Rightarrow listA$ can be defined by:

$$\begin{aligned} app\ A\ (nil\ A')\ l' &\to l' \\ app\ A\ (cons\ A'\ x\ l)\ l' &\to cons\ A\ x\ (app\ A\ x\ l\ l') \\ app\ A\ (app\ A'\ l\ l')\ l'' &\to app\ A\ l\ (app\ A\ l'\ l'') \end{aligned}$$

with $\Gamma = A : \star, x : A, l : listA, l' : listA$ and $\rho = \{A' \mapsto A\}$. Note that the third rule has no counterpart in CIC. Although $app\ A\ (nil\ A')$ is not typable in Γ (since $A' \notin \mathrm{dom}(\Gamma)$), it becomes typable if we apply ρ. This does not matter since, if an instance $app\ A\sigma\ (nil\ A'\sigma)$ is typable then, after the typing rules, $A\sigma$ is convertible to $A'\sigma$. See [6] for details.

We now introduce some restrictions on predicate-level rewrite rules, that generalize usual restrictions of strong elimination. Indeed, it is well known that strong elimination on big inductive types may lead to inconsistencies [9].

Definition 3.6. (Conditions on predicate-level rules)
- For all $F \in \mathcal{F}^{\Box}$, $F\vec{l} \to r \in \mathcal{R}$ and $x \in \mathrm{FV}^{\Box}(r)$, there is κ_x such that $l_{\kappa_x} = x$.
- Predicate-level rules have critical pairs with no rule.

The first condition means that one cannot do matching on predicate arguments, hence that predicate variables are like parameters.

The condition on critical pairs, which is satisfied by CIC recursors, allows us to define an interpretation for defined predicate symbols easily (see Definition 4.3). However, we think that this condition could be weakened. For instance, consider $F : nat \Rightarrow \star \Rightarrow \star \Rightarrow \star$ and the rules:

$$\begin{aligned} F\ 0\ A\ B &\to B \\ F\ (s\ n)\ A\ B &\to A \Rightarrow (F\ n\ A\ B) \end{aligned}$$

$(F\ n\ A\ B)$ is the type of functions with n arguments of type A and output in B. So, it seems reasonable to allow rules derived from inductive consequences of these first two rules, like for instance:

$$F\ (x + y)\ A\ B \to F\ x\ A\ (F\ y\ A\ B)$$

We now prove a simple lemma saying that, for proving a property P for every typing judgment $\Gamma \vdash t : T$, one may proceed by well-founded induction on the symbol precedence and prove that P holds for every typing judgment $\Gamma \vdash_g t : T$ when it holds for every typing judgment $\Gamma \vdash_f t : T$ such that $f < g$.

Lemma 3.1. We have (1) $\Gamma \vdash t : T$ and every symbol occurring in Γ, t, T smaller (resp. strictly smaller) than g if and only if (2) $\Gamma \vdash_g t : T$ (resp. $\Gamma \vdash_g^< t : T$).

Proof:
(1) $\Rightarrow$ (2). One can easily prove by induction on $\Gamma \vdash t : T$ that, (*) if $\Gamma \vdash t : T$ and every symbol occurring in Γ and t is smaller than g, then there exists T' such that $T \to^* T'$ and $\Gamma \vdash_g t : T'$ (see Lemma 54 in [5]). In the (symb) case, it uses the assumption that every symbol occurring in τ_f is strictly smaller than f (Definition 3.4). In the (conv) case, it uses confluence and the assumption that, for every rule $f\vec{l} \to r$, the symbols occurring in r are smaller than f (Definition 3.3). So, assume that $\Gamma \vdash t : T$ and every symbol occurring in Γ, t, T is smaller than g. By (*), there exists T' such that $T \to^* T'$ and $\Gamma \vdash_g t : T'$. By type correctness (Lemma 28 in [5]), either $T = \Box$ or $\Gamma \vdash T : s$. If $T = \Box$ then $T' = T = \Box$ and $\Gamma \vdash_g t : T$. Now, if $\Gamma \vdash T : s$ then, by (*) again, $\Gamma \vdash_g T : s$. Thus, by (conv), $\Gamma \vdash_g t : T$. The same holds with $\vdash_g^<$.
(2) $\Rightarrow$ (1). Easy induction on $\Gamma \vdash_g t : T$. $\Box$

Corollary 3.1. If $\vdash g : \tau_g$ then $\vdash^<_g \tau_g : s_g$.

Proof:
It follows from Lemma 3.1 and the assumption that, for all f, every symbol occurring in τ_f is strictly smaller than f (see Definition 3.4). □

4. Strong normalization

Typed λ-calculi are generally proved strongly normalizing by using Tait and Girard's technique of *reducibility candidates* [14]. The idea of Tait, later extended by Girard to the polymorphic λ-calculus, is to strengthen the induction hypothesis. Instead of proving that every term is strongly normalizable (set $\mathcal{SN}$), one associates to every type T a set $[\![T]\!] \subseteq \mathcal{SN}$, the *interpretation* of T, and proves that every term t of type T is *computable, i.e.* belongs to $[\![T]\!]$. Hereafter, we follow the proof given in [6] which greatly simplifies the one given in [5]. All the definitions and properties of this section are taken from [6].

Definition 4.1. (Reducibility candidates)
We assume given a set $\mathcal{N} \subseteq \mathcal{T}$ of *neutral terms* satisfying the following property: if $t \in \mathcal{N}$ and $u \in \mathcal{T}$ then tu is not head-reducible. We inductively define the complete lattice $\mathcal{R}_t$ of the interpretations for the terms of type t, the ordering $\leq_t$ on $\mathcal{R}_t$, and the greatest element $\top_t \in \mathcal{R}_t$ as follows.

- $\mathcal{R}_t = \{\emptyset\}$, $\leq_t = \subseteq$ and $\top_t = \emptyset$ if $t \neq \Box$ and t is not of the form $(\vec{x} : \vec{T})\star$.
- $\mathcal{R}_s$ is the set of all subsets $R \subseteq \mathcal{T}$ such that:
  (R1) $R \subseteq \mathcal{SN}$ (strong normalization).
  (R2) If $t \in R$ then $\rightarrow(t) = \{t' \in \mathcal{T} \mid t \rightarrow t'\} \subseteq R$ (stability by reduction).
  (R3) If $t \in \mathcal{N}$ and $\rightarrow(t) \subseteq R$ then $t \in R$ (neutral terms).
  Furthermore, $\leq_s = \subseteq$ and $\top_s = \mathcal{SN}$.
- $\mathcal{R}_{(x:U)K}$ is the set of functions R from $\mathcal{T} \times \mathcal{R}_U$ to $\mathcal{R}_K$ such that $R(u, S) = R(u', S)$ whenever $u \rightarrow u'$, $R \leq_{(x:U)K} R'$ iff, for all $(u, S) \in \mathcal{T} \times \mathcal{R}_U$, $R(u, S) \leq_K R'(u, S)$, and $\top_{(x:U)K}(u, S) = \top_K$.

The exact definition of $\mathcal{N}$ is not necessary at this stage. Moreover, the choice of $\mathcal{N}$ may depend on the way predicate symbols are interpreted. The set that we will choose is given in Definition 5.3.

Note that $\mathcal{R}_t = \mathcal{R}_{t'}$ whenever $t \rightarrow t'$ (Lemma 34 in [6]). The proof that $(\mathcal{R}_t, \leq_t)$ is a complete lattice is given in Lemma 35 in [6].

Definition 4.2. (Interpretation schema)
A *candidate assignment* is a function ξ from $\mathcal{X}$ to $\bigcup\{\mathcal{R}_t \mid t \in \mathcal{T}\}$. An assignment ξ *validates* an environment Γ, $\xi \models \Gamma$, if, for all $x \in \mathrm{dom}(\Gamma)$, $x\xi \in \mathcal{R}_{x\Gamma}$. An *interpretation* for a symbol f is an element of $\mathcal{R}_{\tau_f}$. An *interpretation* for a set $\mathcal{G}$ of symbols is a function which, to every symbol $g \in \mathcal{G}$, associates an interpretation for g. The *interpretation* of a term t w.r.t. a candidate assignment ξ, an interpretation I for $\mathcal{F}$ and a substitution θ, is defined by induction on t as follows:

- $[\![t]\!]^I_{\xi,\theta} = \top_t$ if t is an object or a sort,
- $[\![x]\!]^I_{\xi,\theta} = x\xi$,
- $[\![f]\!]^I_{\xi,\theta} = I_f$,
- $[\![(x : U)V]\!]^I_{\xi,\theta} = \{t \in \mathcal{T} \mid \forall u \in [\![U]\!]^I_{\xi,\theta}, \forall S \in \mathcal{R}_U, tu \in [\![V]\!]^I_{\xi^S_x, \theta^u_x}\}$,

- $[\![[x:U]v]\!]^I_{\xi,\theta}(u,S) = [\![v]\!]^I_{\xi^S_x,\theta^u_x}$,
- $[\![tu]\!]^I_{\xi,\theta} = [\![t]\!]^I_{\xi,\theta}(u\theta, [\![u]\!]^I_{\xi,\theta})$,

where $\xi^S_x = \xi \cup \{x \mapsto S\}$ and $\theta^u_x = \theta \cup \{x \mapsto u\}$. A substitution θ is *I-adapted* to a Γ-assignment ξ if $\mathrm{dom}(\theta) \subseteq \mathrm{dom}(\Gamma)$ and, for all $x \in \mathrm{dom}(\theta)$, $x\theta \in [\![x\Gamma]\!]^I_{\xi,\theta}$. A pair (ξ,θ) is (Γ,I)-*valid*, written $\xi,\theta \models_I \Gamma$, if $\xi \models \Gamma$ and θ is I-adapted to ξ. A term t such that $\Gamma \vdash t : T$ is *computable* if, for all (Γ,I)-valid pair (ξ,θ), $t\theta \in [\![T]\!]^I_{\xi,\theta}$. A sub-system $\vdash' \subseteq \vdash$ is *computable* if every term typable in it is computable.

Thanks to the property satisfied by $\mathcal{N}$, one can prove that the interpretation schema defines reducibility candidates: if $\Gamma \vdash t : T$ and $\xi \models \Gamma$, then $[\![t]\!]^I_{\xi,\theta} \in \mathcal{R}_T$ (see Lemma 38 in [6]). Note also that $[\![t]\!]^I_{\xi,\theta} = [\![t]\!]^{I'}_{\xi',\theta'}$ whenever ξ and ξ' agree on the predicate variables free in t, θ and θ' agree on the variables free in t, and I and I' agree on the symbols occurring in t.

Now, the difficult point is to define an interpretation I for every predicate symbol and to prove that every symbol f is computable, *i.e.* $f \in [\![\tau_f]\!]^I$. We define I by induction on the precedence, and simultaneously for the symbols that are in the same equivalence class. We first give the interpretation for defined predicate symbols.

Definition 4.3. (Interpretation of defined predicate symbols)
If every t_i has a normal form t_i^* and $\vec{t}^* = \vec{l}\sigma$ for some rule $F\vec{l} \to r \in \mathcal{R}$, then $I_F(\vec{t},\vec{S}) = [\![r]\!]^I_{\xi,\sigma}$ with $x\xi = S_{\kappa_x}$. Otherwise, $I_F(\vec{t},\vec{S}) = \mathcal{SN}$.

Sufficient conditions of well-definedness are given in [6]. Among other things, it assumes that, for every rule $f\vec{l} \to r$, every symbol occurring in r is smaller than f (see Definition 3.3).

In order for the interpretation to be compatible with the conversion rule, we must make sure that $[\![T]\!]^I_{\xi,\theta} = [\![T']\!]^I_{\xi,\theta}$ whenever $T \to T'$. This property is easily verified if predicate-level rewrite rules have critical pairs with no rule, as required in Definition 3.6 (see Lemma 65 in [6]).

Now, following previous works on inductive types [21, 32], the interpretation of a constant predicate symbol C is defined as the least fixpoint of a monotone function φ_C on the complete lattice $\mathcal{R}_{\tau_C}$. Following Matthes [19], there are essentially two possible definitions that we illustrate by the case of nat. The *introduction-based* definition:

$$\varphi_{nat}(I) = \{t \in \mathcal{SN} \mid t \to^* su \Rightarrow u \in I\}$$

and the *elimination-based* definition:

$$\varphi_{nat}(I) = \{t \in \mathcal{T} \mid \forall (\xi,\theta)\,(\Gamma,I)\text{-valid}, \; rec\,P\theta\,u\theta\,v\theta\,t \in [\![Pn]\!]^I_{\xi,\theta^t_n}\}$$

where $\Gamma = P : nat \Rightarrow \star, u : P0, v : (n : nat)Pn \Rightarrow P(sn)$. In both cases, the monotony of φ_{nat} is ensured by the fact that nat occurs only *positively* in the types of the arguments of its constructors, a common condition for inductive types (for simple types, we say that X occurs positively in $Y \Rightarrow X$ and negatively in $X \Rightarrow Y$). Indeed, Mendler proved that recursors for negative types are not normalizing [21]. Take for instance an inductive type C with constructor $c : (C \Rightarrow nat) \Rightarrow C$. Assume now that we have $p : C \Rightarrow (C \Rightarrow nat)$ defined by the rule $p(cx) \to_{\mathcal{R}} x$. Then, by taking $\omega = [x : C](px)x$, we get the infinite reduction sequence $\omega(c\omega) \to_\beta p(c\omega)(c\omega) \to_{\mathcal{R}} \omega(c\omega) \to_\beta \ldots$ We now extend the notion of positive positions to the terms of CC (in Section 9, we give a more general definition for dealing with inductive-recursive types):

Definition 4.4. (Positive/negative positions)
The sets of *positive positions* $\mathrm{Pos}^+(t)$ and *negative positions* $\mathrm{Pos}^-(t)$ in a term t are inductively defined as follows:

- $\mathrm{Pos}^\delta(s) = \mathrm{Pos}^\delta(x) = \mathrm{Pos}^\delta(f) = \{\varepsilon \mid \delta = +\}$,
- $\mathrm{Pos}^\delta((x:U)V) = 1.\mathrm{Pos}^{-\delta}(U) \cup 2.\mathrm{Pos}^\delta(V)$,
- $\mathrm{Pos}^\delta([x:U]v) = 2.\mathrm{Pos}^\delta(v)$,
- $\mathrm{Pos}^\delta(tu) = 1.\mathrm{Pos}^\delta(t)$,

where ε is the empty word, "." the concatenation, $\delta \in \{-,+\}$, $-+ = -$ and $-- = +$ (usual rules of signs). Moreover, if $\leq$ is an ordering, we let $\leq^+ = \leq$ and $\leq^- = \geq$.

In [6], we used the introduction-based approach since this allowed us to have non-free constructors and pattern-matching on defined symbols, which is forbidden in CIC and does not seem possible with the elimination-based approach. For instance, in CAC, it is possible to formalize the type int of integers by simply taking the symbols $0 : int$, $s : int \Rightarrow int$ and $p : int \Rightarrow int$, together with the rules:

$$\begin{aligned} s\ (p\ x) &\to x \\ p\ (s\ x) &\to x \end{aligned}$$

It is also possible to have the following rule on natural numbers:

$$x \times (y + z) \to (x \times y) + (x \times z)$$

To this end, we considered as constructor not only the usual (constant) constructor symbols but any symbol c whose output type is a constant predicate symbol C (perhaps applied to some arguments). Then, to preserve the monotony of φ_C, matching against c is restricted to the arguments, called *accessible*, in the type of which C occurs only positively. We denote by $\mathrm{Acc}(c)$ the set of accessible arguments of c. For instance, x is accessible in sx since nat occurs only positively in the type of x. But, we also have x and y accessible in $x + y$ since nat occurs only positively in the types of x and y. So, $+$ can be seen as a constructor too, whose arguments are both accessible.

With this approach, we can safely take:

$$\varphi_{nat}(I) = \{t \in \mathcal{SN} \mid \forall f, t \to^* f\vec{u} \Rightarrow \forall j \in \mathrm{Acc}(f), u_j \in [\![U_j]\!]^I_{\xi,\theta}\}$$

where f is any symbol of type $(\vec{y} : \vec{U})nat$ and $\theta = \{\vec{y} \mapsto \vec{u}\}$, whenever an appropriate assignment ξ for the predicate variables of U_j can be defined, which seems possible only if the condition (I6) is satisfied (see Definition 1.1). Here, since nat has no parameter, this condition is satisfied only if U_j has no predicate argument.

As a consequence, if $f\vec{t}$ is computable then, for all $j \in \mathrm{Acc}(f)$, t_j is computable (see Lemma 53 in [6]). This means that, when a rule applies, the matching substitution σ is computable. This property is then used for proving the termination of higher-order rewrite rules by using the notion of computability closure of a rule left hand-side (see Definition 25 in [6]). The computability closure is defined in such a way that, if r is in the computability closure of $f\vec{l}$ then, for all computable substitution σ, $r\sigma$ is computable whenever the terms in $\vec{l}\sigma$ are computable (see Theorem 67 in [6]).

As for first-order rewrite rules, *i.e.* rules with algebraic right hand-sides and variables of first-order data type only, it is well known since the pioneering works of Breazu-Tannen and Gallier [7], and Okada

[24], that their combination with non-dependent typed λ-calculi preserves strong normalization. It comes from the fact that first-order rewriting cannot create new β-redexes. This result can be extended to our more general framework if the following two conditions are satisfied:

- Since we consider the combination of a set of first-order rewrite rules and a set of higher-order rewrite rules, and since strong normalization is not modular [30], we require first-order rewrite rules to be non duplicating (no variable occurs more times in a right hand-side than in a left hand-side) [28, 17].
- For proving that first-order rewrite rules preserve not only strong normalization but also computability, we must make sure that, for first-order data types, computability is equivalent to strong normalization.

In fact, we consider a slightly more general notion of first-order data type than usual: our first-order data types can be dependent if the dependencies are first-order data types too (*e.g.* lists of natural numbers of fixed length).

Definition 4.5. (First-order data types)
Types equivalent to C are *first-order data types*[3] if, for all $D \simeq C$, $D : (\vec{z} : \vec{V})\star$, $\{\vec{z}\} \subseteq \mathcal{X}^\star$ and, for all $d : (\vec{x} : \vec{T})D\vec{v}$, $\{\vec{x}\} \subseteq \mathcal{X}^\star$, $\mathrm{Acc}(d) = \{1, \ldots, |\vec{x}|\}$ and every T_j is of the form $E\vec{w}$ with $E \leq C$ a first-order data type too.

5. Abstract recursors

From now on, we assume that the set of constant predicate symbols $\mathcal{CF}^\Box$ is divided in two disjoint sets: the set $\mathcal{CF}^\Box_{intro}$ of predicate symbols interpreted by the introduction-based method of [6], and the set $\mathcal{CF}^\Box_{elim}$ of predicate symbols interpreted by the elimination-based method of the present paper.

We now introduce an abstract notion of recursor for dealing with the elimination-based method in a general way.

Definition 5.1. (Pre-recursors)
A *pre-recursor* for a symbol $C : (\vec{z} : \vec{V})\star$ in $\mathcal{CF}^\Box_{elim}$ is any symbol $f \notin \mathcal{C}ons$ such that:

- τ_f is of the form $(\vec{z} : \vec{V})(z : C\vec{z})W$,
- every predicate symbol occurring in W is smaller than C,
- every rule defining f is of the form $f\vec{z}(c\vec{t})\vec{u} \to r$ with c constant, $\vec{z} \in \mathcal{X}$ and $\mathrm{FV}(r) \cap \{\vec{z}\} = \emptyset$,

The form of a pre-recursor type may seem restrictive. However, since termination is not established yet, we cannot consider the normal form of a type when testing if it matches some given form. Moreover, in an environment, every two variables whose types do not depend on each other can be permuted without modifying the set of terms typable in this environment (see Lemma 18 in [5]). So, our results also apply on symbols whose type can be brought to this form by various applications of this lemma.

Definition 5.2. (Positivity conditions)
A pre-recursor $f : (\vec{z} : \vec{V})(z : C\vec{z})W$ is a *recursor* if it satisfies the following *positivity conditions*:[4]

- no defined predicate $F \simeq C$ occurs in W: $\mathrm{Pos}(F, W) = \emptyset$,

[3] Called *primitive* in [6].
[4] In Section 9, we give weaker conditions for dealing with inductive-recursive types.

– every constant predicate $D \simeq C$ occurs only positively in W: $\mathrm{Pos}(D, W) \subseteq \mathrm{Pos}^+(W)$.

A recursor f of sort $s_f = \star$ (resp. $\Box$) is *weak* (resp. *strong*). We assume that every type $C \in \mathcal{CF}^{\Box}_{elim}$ has a non empty set $\mathcal{R}ec(C)$ of recursors, and that $\mathcal{R}ec(C) \cap \mathcal{R}ec(D) = \emptyset$ whenever C and D are two distinct predicate symbols of $\mathcal{CF}^{\Box}_{elim}$.

We now define a set $\mathcal{N}$ of *neutral terms* (see Definition 4.1) that is adapted to both the introduction-based and the elimination-based approach.

Definition 5.3. (Neutral terms)
For the set $\mathcal{N}$ of *neutral terms* (see Definition 4.1), we choose the set of all terms not of the form:

– abstraction: $[x : T]u$,
– partial application: $f\vec{t}$ with f defined by some rule $f\vec{l} \to r$ with $|\vec{l}| > |\vec{t}|$,
– constructor: $f\vec{t}$ with $\tau_f = (\vec{y} : \vec{U})C\vec{v}$, $|\vec{t}| = |\vec{y}|$, $C \in \mathcal{CF}^{\Box}$, and f constant whenever $C \in \mathcal{CF}^{\Box}_{elim}$.

In comparison with Definition 31 in [6], we just added the restriction, in the constructor case, that f is constant if $C \in \mathcal{CF}^{\Box}_{elim}$. This therefore changes nothing if $C \in \mathcal{CF}^{\Box}_{intro}$.

We now define the interpretation of the equivalence class of a symbol $C \in \mathcal{CF}^{\Box}_{elim}$. Since we proceed by induction on the precedence for defining the interpretation of predicate symbols, we can assume that an interpretation for the symbols strictly smaller than C is already defined. The set of interpretations for constant predicate symbols equivalent to C, ordered point-wise, is a complete lattice. We now define the monotone function φ on this lattice whose fixpoint will be the interpretation for constant predicate symbols equivalent to C.

Definition 5.4. (Interpretation of constant predicate symbols from $\mathcal{CF}^{\Box}_{elim}$)
If every t_i has a normal form t_i^* then $\varphi^I_C(\vec{t}, \vec{S})$ is the set of terms t such that, for all $f \in \mathcal{R}ec(C)$ of type $(\vec{z} : \vec{V})(z : C\vec{z})(\vec{y} : \vec{U})V$ with V not a product, and for all $\vec{y}\xi$ and $\vec{y}\theta$, if $\xi^{\vec{S}}_{\vec{z}}, \theta^{\vec{t}\,t}_{\vec{z}z} \models_I \vec{y} : \vec{U}$ then $f\vec{t}^*t\vec{y}\theta \in [\![V]\!]^I_{\xi^{\vec{S}}_{\vec{z}}, \theta^{\vec{t}\,t}_{\vec{z}z}}$. Otherwise, $\varphi^I_C(\vec{t}, \vec{S}) = \mathcal{SN}$.

This interpretation is well defined since, by Definition 5.1, every predicate symbol occurring in $(\vec{y} : \vec{U})V$ is smaller than C. Furthermore, one can easily check that φ^I_C is stable by reduction: if $\vec{t} \to \vec{t}'$ then $\varphi^I_C(\vec{t}, \vec{S}) = \varphi^I_C(\vec{t}', \vec{S})$. We now prove that $\varphi^I_C(\vec{t}, \vec{S})$ is a reducibility candidate.

Lemma 5.1. $R = \varphi^I_C(\vec{t}, \vec{S})$ is a reducibility candidate.

Proof:

(R1) Let $t \in R$. We must prove that $t \in \mathcal{SN}$. Since $\mathcal{R}ec(C) \neq \emptyset$, there is at least one recursor f. Take $y_i\theta = y_i$ and $y_i\xi = \top_{U_i}$. We clearly have $\xi^{\vec{S}}_{\vec{z}}, \theta^{\vec{t}\,t}_{\vec{z}z} \models_I \vec{y} : \vec{U}$. Therefore, $f\vec{t}^*t\vec{y} \in S = [\![V]\!]^I_{\xi^{\vec{S}}_{\vec{z}}, \theta^{\vec{t}\,t}_{\vec{z}z}}$. Now, since S satisfies (R1), $f\vec{t}^*t\vec{y} \in \mathcal{SN}$ and $t \in \mathcal{SN}$.

(R2) Let $t \in R$ and $t' \in {\to}(t)$. We must prove that $t' \in R$, hence that $f\vec{t}^*t'\vec{y}\theta \in S = [\![V]\!]^I_{\xi^{\vec{S}}_{\vec{z}}, \theta^{\vec{t}\,t}_{\vec{z}z}}$. This follows from the fact that $f\vec{t}^*t\vec{y}\theta \in S$ (since $t \in R$) and S satisfies (R2).

(R3) Let t be a neutral term such that $\to(t) \subseteq R$. We must prove that $t \in R$, hence that $u = f\vec{t}^*t\vec{y}\theta \in S = [\![V]\!]^I_{\xi^{\vec{S}}_{\vec{z}},\theta^{\vec{t}\,t}_{\vec{z}z}}$. Since u is neutral and S satisfies (R3), it suffices to prove that $\to(u) \subseteq S$. Since $\vec{y}\theta \in \mathcal{SN}$ by (R1), we proceed by induction on $\vec{y}\theta$ with $\to$ as well-founded ordering. The only difficult case could be when u is head-reducible, but this is not possible since t is neutral.

□

The fact that φ is monotone, hence has a least fixpoint, follows from the positivity conditions.

Lemma 5.2. Let $I \leq_f I'$ iff $I_f \leq I'_f$ and, for all $g \neq f$, $I_g = I'_g$. If $I \leq_f I'$, $\mathrm{Pos}(f,t) \subseteq \mathrm{Pos}^\delta(t)$, $\Gamma \vdash t : T$ and $\xi \models \Gamma$ then $[\![t]\!]^I_{\xi,\theta} \leq^\delta [\![t]\!]^{I'}_{\xi,\theta}$.

Proof:
By induction on t.

- $[\![s]\!]^I_{\xi,\theta} = \top_s = [\![s]\!]^{I'}_{\xi,\theta}$.
- $[\![x]\!]^I_{\xi,\theta} = x\xi = [\![x]\!]^{I'}_{\xi,\theta}$.
- Let $R = [\![g\vec{t}]\!]^I_{\xi,\theta}$ and $R' = [\![g\vec{t}]\!]^{I'}_{\xi,\theta}$. $R = I_g(\vec{t}\theta, \vec{S})$ with $\vec{S} = [\![\vec{t}]\!]^I_{\xi,\theta}$. $R' = I'_g(\vec{t}\theta, \vec{S}')$ with $\vec{S}' = [\![\vec{t}]\!]^{I'}_{\xi,\theta}$. Since $\mathrm{Pos}(f,\vec{t}) = \emptyset$, $\vec{S} = \vec{S}'$. Now, if $f = g$ then $R \leq R'$ and $\delta = +$ necessarily. Otherwise, $R = R'$.
- Let $R = [\![(x : U)V]\!]^I_{\xi,\theta}$ and $R' = [\![(x : U)V]\!]^{I'}_{\xi,\theta}$. $R = \{t \in \mathcal{T} \mid \forall u \in [\![U]\!]^I_{\xi,\theta}, \forall S \in \mathcal{R}_U, tu \in [\![V]\!]^I_{\xi^S_x,\theta^u_x}\}$. $R' = \{t \in \mathcal{T} \mid \forall u \in [\![U]\!]^{I'}_{\xi,\theta}, \forall S \in \mathcal{R}_U, tu \in [\![V]\!]^I_{\xi'^S_x,\theta^u_x}\}$. Since $\mathrm{Pos}^\delta((x : U)V) = 1.\mathrm{Pos}^{-\delta}(U) \cup 2.\mathrm{Pos}^\delta(V)$, $\mathrm{Pos}(f,U) \subseteq \mathrm{Pos}^{-\delta}(U)$ and $\mathrm{Pos}(f,V) \subseteq \mathrm{Pos}^\delta(V)$. Therefore, by induction hypothesis, $[\![U]\!]^I_{\xi,\theta} \leq^{-\delta} [\![U]\!]^{I'}_{\xi,\theta}$ and $[\![V]\!]^I_{\xi^S_x,\theta^u_x} \leq^\delta [\![V]\!]^I_{\xi'^S_x,\theta^u_x}$. So, $R \leq^\delta R'$. Indeed, if $\delta = +$, $t \in R$ and $u \in [\![U]\!]^{I'}_{\xi,\theta} \subseteq [\![U]\!]^I_{\xi,\theta}$ then $tu \in [\![V]\!]^I_{\xi^S_x,\theta^u_x} \subseteq [\![V]\!]^I_{\xi'^S_x,\theta^u_x}$ and $t \in R'$. If $\delta = -$, $t \in R'$ and $u \in [\![U]\!]^I_{\xi,\theta} \subseteq [\![U]\!]^{I'}_{\xi,\theta}$ then $tu \in [\![V]\!]^I_{\xi'^S_x,\theta^u_x} \subseteq [\![V]\!]^I_{\xi^S_x,\theta^u_x}$ and $t \in R$.
- Let $R = [\![[x : U]v]\!]^I_{\xi,\theta}$ and $R' = [\![[x : U]v]\!]^{I'}_{\xi,\theta}$. R and R' have the same domain $\mathcal{T} \times \mathcal{R}_U$ and the same codomain $\mathcal{R}_V$. $R(u,S) = [\![v]\!]^I_{\xi^S_x,\theta^u_x}$ and $R'(u,S) = [\![v]\!]^I_{\xi'^S_x,\theta^u_x}$. Since $\mathrm{Pos}^\delta([x : U]v) = 2.\mathrm{Pos}^\delta(v)$, $\mathrm{Pos}(f,v) \subseteq \mathrm{Pos}^\delta(v)$. Therefore, by induction hypothesis, $R(u,S) \leq^\delta R'(u,S)$ and $R \leq^\delta R'$.
- Let $R = [\![tu]\!]^I_{\xi,\theta}$ and $R' = [\![tu]\!]^{I'}_{\xi,\theta}$. $R = [\![t]\!]^I_{\xi,\theta}(u\theta, S)$ with $S = [\![u]\!]^I_{\xi,\theta}$. $R' = [\![t]\!]^{I'}_{\xi,\theta}(u\theta, S')$ with $S' = [\![u]\!]^{I'}_{\xi,\theta}$. Since $\mathrm{Pos}^\delta(tu) = 1.\mathrm{Pos}^\delta(t)$, $\mathrm{Pos}(f,t) \subseteq \mathrm{Pos}^\delta(t)$ and $\mathrm{Pos}(f,u) = \emptyset$. Therefore, $S = S'$ and, by induction hypothesis, $[\![t]\!]^I_{\xi,\theta} \leq^\delta [\![t]\!]^{I'}_{\xi,\theta}$. So, $R \leq^\delta R'$.

□

Lemma 5.3. φ is monotone.

Proof:
Let $I \leq J$. We must prove that, for all C, $\vec{t}$, $\vec{S}$, $\varphi^I_C(\vec{t},\vec{S}) \subseteq \varphi^J_C(\vec{t},\vec{S})$. If some t_i has no normal form then $\varphi^I_C(\vec{t},\vec{S}) = \varphi^J_C(\vec{t},\vec{S}) = \mathcal{SN}$. Assume now that every t_i has a normal form t^*_i. Let $t \in \varphi^I_C(\vec{t},\vec{S})$, $f \in \mathcal{R}ec(C)$ with $\tau_f = (\vec{z} : \vec{V})(z : C\vec{z})(\vec{y} : \vec{U})V$, $\vec{y}\xi$ and $\vec{y}\theta$ such that $\xi^{\vec{S}}_{\vec{z}}, \theta^{\vec{t}\,t}_{\vec{z}z} \models_J \vec{y} : \vec{U}$. We must prove that $f\vec{t}^*t\vec{y}\theta \in [\![V]\!]^J_{\xi^{\vec{S}}_{\vec{z}},\theta^{\vec{t}\,t}_{\vec{z}z}}$. $\xi^{\vec{S}}_{\vec{z}}, \theta^{\vec{t}\,t}_{\vec{z}z} \models_J \vec{y} : \vec{U}$ means that $\vec{y}\theta \in [\![\vec{U}]\!]^J_{\xi^{\vec{S}}_{\vec{z}},\theta^{\vec{t}\,t}_{\vec{z}z}}$.

Let $W = (\vec{y} : \vec{U})V$. By assumption, for every $D \simeq C$, $\mathrm{Pos}(D, W) \subseteq \mathrm{Pos}^+(W)$. Thus, $\mathrm{Pos}(D, \vec{U}) \subseteq \mathrm{Pos}^-(\vec{U})$ and $\mathrm{Pos}(D, V) \subseteq \mathrm{Pos}^+(V)$. Hence, by Lemma 5.2, $\xi_{\vec{z}}^{\vec{S}}, \theta_{\vec{z}z}^{\vec{t}t} \models_I \vec{y} : \vec{U}$ and $[\![V]\!]^I_{\xi_{\vec{z}}^{\vec{S}}, \theta_{\vec{z}z}^{\vec{t}t}} \subseteq [\![V]\!]^J_{\xi_{\vec{z}}^{\vec{S}}, \theta_{\vec{z}z}^{\vec{t}t}}$. Thus, $f\vec{t^*}t\vec{y}\theta \in [\![V]\!]^J_{\xi_{\vec{z}}^{\vec{S}}, \theta_{\vec{z}z}^{\vec{t}t}}$. □

6. Admissible recursors

Now, for getting termination of $\beta \cup \mathcal{R}$, we need to prove that every symbol f is computable, *i.e.* $f \in [\![\tau_f]\!]$. To this end, we give general conditions on recursors. We focus on what is new and refer the reader to [6] for the other cases. After Lemma 3.1, we know that we can proceed by induction on the precedence for proving the computability of well-typed terms. So, when defining conditions on a symbol f, we can always assume w.l.o.g. that $\vdash_f^<$ is computable, *i.e.* terms with symbols strictly smaller than f are computable (see Definition 4.2). In particular, every subterm of τ_f is computable (see Corollary 3.1).

Definition 6.1. (Admissible recursors)
Let $C : (\vec{z} : \vec{V})\star$ be a constant predicate symbol such that $\mathcal{R}ec(C) \neq \emptyset$. We assume that every symbol $c : (\vec{x} : \vec{T})C\vec{v}$ is equipped with a set $\mathrm{Acc}(c) \subseteq \{1, \ldots, |\vec{x}|\}$ of *accessible arguments*. A *constructor* of C is any constant symbol $c : (\vec{x} : \vec{T})C\vec{v}$.

The set $\mathcal{R}ec(C)$ is *complete w.r.t. accessibility* if, for all constructor $c : (\vec{x} : \vec{T})C\vec{v}$, $j \in \mathrm{Acc}(c)$, $\vec{x}\eta$ and $\vec{x}\sigma$, if $\eta \models \Gamma_c$, $\vec{v}\sigma \in \mathcal{SN}$ and $c\vec{x}\sigma \in [\![C\vec{v}]\!]_{\eta,\sigma}$ then $x_j\sigma \in [\![T_j]\!]_{\eta,\sigma}$.

A recursor $f : (\vec{z} : \vec{V})(z : C\vec{z})(\vec{y} : \vec{U})V$ is *head-computable w.r.t.* a constructor $c : (\vec{x} : \vec{T})C\vec{v}$ if, whenever $\vdash_f^<$ is computable, for all $\vec{x}\eta$, $\vec{x}\sigma$, $\vec{y}\xi$, $\vec{y}\theta$, $\vec{S} = [\![\vec{v}]\!]_{\eta,\sigma}$ such that $\eta, \sigma \models \Gamma_c$ and $\xi_{\vec{z}}^{\vec{S}}, \theta_{\vec{z}}^{\vec{v}\sigma}{}_z^{c\vec{x}\sigma} \models \vec{y} : \vec{U}$, every head-reduct of $f\vec{v}\sigma(c\vec{x}\sigma)\vec{y}\theta$ belongs to $[\![V]\!]_{\xi_{\vec{z}}^{\vec{S}}, \theta_{\vec{z}}^{\vec{v}\sigma}{}_z^{c\vec{x}\sigma}}$. A recursor is *head-computable* if it is head-computable w.r.t. every constructor of C. $\mathcal{R}ec(C)$ is *head-computable* if all its recursors are head-computable.

$\mathcal{R}ec(C)$ is *admissible* if it is head-computable and complete w.r.t. accessibility.

Completeness w.r.t. accessibility exactly insures that, if $c\vec{t}$ is computable then, for all $j \in \mathrm{Acc}(c)$, t_j is computable (Lemma 53 in [6]), hence that non-recursor higher-order symbols are computable (see Lemma 68 in [6]). We now prove that the elimination-based interpretation of first-order data types is $\mathcal{SN}$, hence that first-order symbols are computable (see Lemma 63 in [6]).

Lemma 6.1. If C is a first-order data type and $\mathcal{R}ec(C)$ is head-computable then $I_C(\vec{t}, \vec{S}) = \mathcal{SN}$.

Proof:
First note that $S_i = \emptyset$ since $\{\vec{z}\} \subseteq \mathcal{X}^\star$. So, we do not write $\vec{S}$ in the following. By definition, for all $\vec{t}$, $I_C(\vec{t}) \subseteq \mathcal{SN}$. We now prove that, if $t \in \mathcal{SN}$ then, for all $\vec{t}$, $t \in I_C(\vec{t})$, by induction on t with $\rightarrow \cup \rhd$ as well-founded ordering. If some t_i has no normal form then $t \in I_C(\vec{t}) = \mathcal{SN}$. Assume now that every t_i has a normal form t_i^*. Let $f : (z : C)(\vec{y} : \vec{U})V$ be a recursor of C, $\vec{y}\xi$, $\vec{y}\theta$ and $\sigma = \theta_{\vec{z}z}^{\vec{t}t}$ such that $\xi, \sigma \models \vec{y} : \vec{U}$. We must prove that $v = f\vec{t^*}t\vec{y}\theta \in S = [\![V]\!]_{\xi,\sigma}$. Since v is neutral, it suffices to prove that $\rightarrow(v) \subseteq S$. We proceed by induction on $t\vec{y}\theta$ with $\rightarrow$ as well-founded ordering ($\vec{y}\theta \in \mathcal{SN}$ by R1). If the reduction takes place in $t\vec{y}\theta$, we can conclude by induction hypothesis. Assume now that v' is a head-reduct of v. By assumption on recursors, t is of the form $c\vec{u}$ with $c : (\vec{x} : \vec{T})C\vec{v}$. Let $\gamma = \{\vec{x} \mapsto \vec{u}\}$.

Since C is a first-order data type, every u_j is accessible and every T_j is of the form $D\vec{w}$ with D a first-order data type too. Thus, by induction hypothesis, for all j, $u_j \in I_D(\vec{w}\gamma)$. Therefore, $\emptyset, \gamma \models \Gamma_c$ and $v' \in S$ since $\xi, \sigma \models \vec{y} : \vec{U}$ and recursors are assumed to be head-computable. □

Lemma 6.2. Head-computable recursors are computable.

Proof:
Let $f : (\vec{z} : \vec{V})(z : C\vec{z})(\vec{y} : \vec{U})V$ be a recursor and assume that $\xi, \theta \models \Gamma_f$. We must prove that $v = f\vec{z}\theta z\theta\vec{y}\theta \in S = [\![V]\!]_{\xi,\theta}$. Since v is neutral, it suffices to prove that $\rightarrow(v) \subseteq S$. We proceed by induction on $\vec{z}\theta z\theta\vec{y}\theta$ with $\rightarrow$ as well-founded ordering ($\vec{z}\theta z\theta\vec{y}\theta \in \mathcal{SN}$ by R1). If the reduction takes place in $\vec{z}\theta z\theta\vec{y}\theta$, we conclude by induction hypothesis. Assume now that we have a head-reduct v'. By definition of recursors (see Definition 5.1), $z\theta$ is of the form $c\vec{u}$ with $c : (\vec{x} : \vec{T})C\vec{v}$, and v' is also a head-reduct of $v_0 = f(\vec{z}\theta)^* z\theta\vec{y}\theta$. Since $\xi, \theta \models \Gamma_f$, we have $z\theta = c\vec{u} \in [\![C\vec{z}]\!]_{\xi,\theta} = I_C(\vec{z}\theta, \vec{z}\xi)$. Therefore, by definition of I_C, $v_0 \in S$ and, by (R2), $v' \in S$. □

Lemma 6.3. (Computability)
For all g, if $\vdash_g^<$ is computable then $\vdash_g$ is computable.

Proof:
We prove that, if $\Gamma \vdash_g t : T$ and $\eta, \sigma \models \Gamma$ then $t\sigma \in [\![T]\!]_{\eta,\sigma}$, by induction on $\Gamma \vdash_g t : T$. We only detail the (symb) case. The other cases are detailed in Lemma 66 in [6]. So, assume that $\vdash_g f : \tau_f$. If $f < g$ then, by Lemma 3.1, $\vdash_g^< f : \tau_f$ and f is computable since $\vdash_g^<$ is assumed to be computable. Otherwise, $f \simeq g$ and $\vdash_f^< = \vdash_g^<$. If f is a recursor then we can conclude by Lemma 6.2. So, assume that f is not a recursor and that $\tau_f = (\vec{x} : \vec{T})U$ with U distinct from a product. By Definition 4.2, f is computable iff, for all Γ_f-valid pair (η, σ), $t = f\vec{x}\sigma \in R = [\![U]\!]_{\eta,\sigma}$.

If t is neutral then, by definition 4.1, it suffices to prove that $\rightarrow(t) \subseteq R$, which follows from Lemmas 63 and 68 in [6]. Assume now that t is not neutral. Then, $U = C\vec{v}$ with $C \in \mathcal{CF}^{\Box}$, and $R = I_C(\vec{v}\sigma, \vec{S})$ with $\vec{S} = [\![\vec{v}]\!]_{\eta,\sigma}$. If $C \in \mathcal{CF}^{\Box}_{intro}$ then, again, it follows from Lemmas 63 and 68 in [6]. Otherwise, $C \in \mathcal{CF}^{\Box}_{elim}$ and, by Definition 5.1, f is constant.

By Corollary 3.1, $\vdash_f^< \tau_f : s_f$. Since, by assumption, $\vdash_f^<$ is computable, by (R1), $\vec{v}\sigma \in \mathcal{SN}$. So, let $g : (\vec{z} : \vec{V})(z : C\vec{z})(\vec{y} : \vec{U})V$ be a recursor of C, $\vec{y}\xi$ and $\vec{y}\theta$ such that $\xi^{\vec{S}}_{\vec{z}}, \theta^{\vec{v}\sigma\, f\vec{x}\sigma}_{\vec{z}\ z} \models \vec{y} : \vec{U}$. We must prove that $v = g(\vec{v}\sigma)^*(f\vec{x}\sigma)\vec{y}\theta \in S = [\![V]\!]_{\xi^{\vec{S}}_{\vec{z}}, \theta^{\vec{v}\sigma\, f\vec{x}\sigma}_{\vec{z}\ z}}$. Since v is neutral, it suffices to prove that $\rightarrow(v) \subseteq S$. By (R1), $\vec{x}\sigma\vec{y}\theta \in \mathcal{SN}$. So, we can proceed by induction on $\vec{x}\sigma\vec{y}\theta$ with $\rightarrow$ as well-founded ordering. No reduction can take place at the top of $f\vec{x}\sigma$ since f is constant. In the case of a reduction in $\vec{x}\sigma\vec{y}\theta$, we conclude by induction hypothesis. Finally, in the case of a head-reduction, we conclude by head-computability of g. □

We can now state our main result:

Theorem 6.1. (Strong normalization)
$\beta \cup \mathcal{R}$ preserves typing and is strongly normalizing if:

- $\beta \cup \mathcal{R}$ is confluent[5] (if there are predicate-level rules),
- rewrite rules are well-typed,

[5] Again, this is the case if, for instance, $\mathcal{R}$ is confluent and left-linear [22].

– every constant predicate symbol $C \in \mathcal{CF}^{\Box}_{elim}$ is equipped with an admissible set $\mathcal{R}ec(C)$ of recursors,
– strong recursors and non-recursor symbols satisfy the conditions given in Definition 29 in [6].

Proof:
After Lemma 3.1, we can proceed by induction on the precedence. Hence, by Lemma 6.3, every well-typed term is computable. Let t be a term such that $\Gamma \vdash t : T$. With $x\theta = x$ and $x\xi = \top_{x\Gamma}$, we clearly have $\xi, \theta \models \Gamma$ since, by Lemma 33 in [6], variables are elements of every candidate. Thus, by (R1), $t \in \mathcal{SN}$. □

As an application example of this theorem, we prove just below the admissibility of a large class of recursors for strictly positive types, from which Coq's recursors [8] can be easily derived (see Section 7). Before that, let us remark that the condition I6 and the safeness condition described in the introduction (Definitions 1.1 and 1.2 respectively) are not necessary anymore for weak recursors. On the other hand, the safeness condition is still necessary for non-recursor symbols and strong recursors on types like $JMeq$.

Definition 6.2. (Canonical recursors for strictly positive types)
Let $C : (\vec{z} : \vec{V})\star$ and $\vec{c}$ be *strictly positive* constructors of C, that is, if c_i is of type $(\vec{x} : \vec{T})C\vec{v}$ then either no type equivalent to C occurs in T_j or T_j is of the form $(\vec{\alpha} : \vec{W})C\vec{w}$ with no type equivalent to C in $\vec{W}$. The *parameters* of C are the biggest sequence $\vec{q}$ such that $C : (\vec{q} : \vec{Q})(\vec{z} : \vec{V})\star$ and each c_i is of type $(\vec{q} : \vec{Q})(\vec{x} : \vec{T})C\vec{q}\vec{v}$ with $T_j = (\vec{\alpha} : \vec{W})C\vec{q}\vec{w}$ if C occurs in T_j.

The *canonical weak recursor* of C w.r.t. $\vec{c}$ is $rec^{\star}_{\vec{c}} : (\vec{q} : \vec{Q})(\vec{z} : \vec{V})(z : C\vec{q}\vec{z})(P : (\vec{z} : \vec{V})C\vec{q}\vec{z} \Rightarrow \star)(\vec{y} : \vec{U})P\vec{z}z$ with $U_i = (\vec{x} : \vec{T})(\vec{x}' : \vec{T}')P\vec{v}(c_i\vec{q}\vec{x})$, $T'_j = (\vec{\alpha} : \vec{W})P\vec{w}(x_j\vec{\alpha})$ if $T_j = (\vec{\alpha} : \vec{W})C\vec{q}\vec{w}$, and $T'_j = T_j$ otherwise, defined by the rules $rec^{\star}_{\vec{c}}\vec{q}\vec{z}(c_i\vec{q}'\vec{x})P\vec{y} \to y_i\vec{x}\vec{t}'$ where $\vec{q}, \vec{z}, \vec{q}', \vec{x}, P, \vec{y}$ are variables, $t'_j = [\vec{\alpha} : \vec{W}](rec^{\star}_{\vec{c}}\vec{q}\vec{w}(x_j\vec{\alpha})P\vec{y})$ if $T_j = (\vec{\alpha} : \vec{W})C\vec{q}\vec{w}$, and $t'_j = x_j$ otherwise.[6]

The *canonical strong recursor*[7] of C w.r.t. $\vec{c}$ and $P = [\vec{z} : \vec{V}][z : C\vec{q}\vec{z}]Q$ is $rec^{P}_{\vec{c}} : (\vec{q} : \vec{Q})(\vec{z} : \vec{V})(z : C\vec{q}\vec{z})(\vec{y} : \vec{U})Q$ with $U_i = (\vec{x} : \vec{T})(\vec{x}' : \vec{T}')Q\{\vec{z} \mapsto \vec{v}, z \mapsto c_i\vec{q}\vec{x}\}$, $T'_j = (\vec{\alpha} : \vec{W})Q\{\vec{z} \mapsto \vec{w}, z \mapsto x_j\vec{\alpha}\}$ if $T_j = (\vec{\alpha} : \vec{W})C\vec{q}\vec{w}$, and $T'_j = T_j$ otherwise, defined by the rules $rec^{P}_{\vec{c}}\,\vec{q}\vec{z}(c_i\vec{q}'\vec{x})\vec{y} \to y_i\vec{x}\vec{t}'$ where $\vec{q}, \vec{z}, \vec{q}', \vec{x}, \vec{y}$ are variables, $t'_j = [\vec{\alpha} : \vec{W}](rec^{P}_{\vec{c}}\,\vec{q}\vec{w}(x_j\vec{\alpha})\vec{y})$ if $T_j = (\vec{\alpha} : \vec{W})C\vec{q}\vec{w}$, and $t'_j = x_j$ otherwise.

Lemma 6.4. The rules defining canonical recursors preserve typing.

Proof:
For the rule $rec^{\star}_{\vec{c}}\vec{q}\vec{z}(c_i\vec{q}'\vec{x})P\vec{y} \to y_i\vec{x}\vec{t}'$, take $\Gamma = \vec{q} : \vec{Q}, \vec{x} : \vec{T}, P : (\vec{z} : \vec{V})C\vec{q}\vec{z} \Rightarrow \star, \vec{y} : \vec{U}$ and $\rho = \{\vec{z} \mapsto \vec{v}, \vec{q}' \mapsto \vec{q}\}$. We prove the conditions required in Section 3:

– One can easily check that $\Gamma \vdash y_i\vec{x}\vec{t}' : P\vec{v}(c_i\vec{q}\vec{x})$.
– Assume now that $\Delta \vdash (rec^{\star}_{\vec{c}}\vec{q}\vec{z}(c_i\vec{q}'\vec{x})P\vec{y})\sigma : T$. We must prove that $\sigma : \Gamma \rightsquigarrow \Delta$ and $\sigma \downarrow \rho\sigma$. Both properties follow by inversion of the typing judgment and confluence.

The proof is about the same for strong recursors. □

Lemma 6.5. The set of canonical recursors is complete w.r.t. accessibility.[8]

[6] We could erase the useless arguments $t'_j = x_j$ when $T'_j = T_j$ as it is done in CIC.
[7] Strong recursors cannot be defined exactly like weak recursors by simply taking $P : (\vec{z} : \vec{V})C\vec{q}\vec{z} \Rightarrow \Box$ since $(\vec{z} : \vec{V})C\vec{q}\vec{z} \Rightarrow \Box$ is not typable in CC. They must be defined for each P. That is why Werner considered a slightly more general PTS in [32].
[8] In [32] (Lemma 4.35), Werner proves a similar result.

Proof:
Let $c = c_i : (\vec{q} : \vec{Q})(\vec{x} : \vec{T})C\vec{q}\vec{v}$ be a constructor of $C : (\vec{q} : \vec{Q})(\vec{z} : \vec{V})\star$, $\vec{q}\eta$, $\vec{x}\eta$, $\vec{q}\sigma$ and $\vec{x}\sigma$ such that $\vec{q}\sigma\vec{v}\sigma \in \mathcal{SN}$ and $c\vec{q}\sigma\vec{x}\sigma \in [\![C\vec{q}\vec{v}]\!]_{\eta,\sigma} = I_C(\vec{q}\sigma\vec{v}\sigma, \vec{q}\eta[\![\vec{v}]\!]_{\eta,\sigma})$. Let $\vec{a} = \vec{q}\vec{x}$ and $\vec{A} = \vec{Q}\vec{T}$. We must prove that, for all j, $a_j\sigma \in [\![A_j]\!]_{\eta,\sigma}$. For the sake of simplicity, we assume that weak and strong recursors have the same syntax. Since $\vec{q}\sigma\vec{v}\sigma$ have normal forms, it suffices to find P and u such that $rec_c\vec{q}\vec{v}(c\vec{a})Pu \to u\vec{x}\vec{t} \to_\beta^* a_j$. Take $P = [\vec{z} : \vec{V}][z : C\vec{q}\vec{z}]A_j$ and $u = [\vec{x} : \vec{T}][\vec{x}' : \vec{T}']a_j$. □

Lemma 6.6. Canonical recursors are head-computable.

Proof:
Let $f = rec^\star : (\vec{q} : \vec{Q})(\vec{z} : \vec{V})(z : C\vec{q}\vec{z})(P : (\vec{z} : \vec{V})C\vec{q}\vec{z} \Rightarrow \star)(\vec{y} : \vec{U})P\vec{z}z$ be the canonical weak recursor w.r.t. $\vec{c}$, $T = (\vec{z} : \vec{V})C\vec{q}\vec{z} \Rightarrow \star$, $c = c_i : (\vec{q} : \vec{Q})(\vec{x} : \vec{T})C\vec{q}\vec{v}$, $\vec{q}\eta$, $\vec{q}\sigma$, $\vec{x}\eta$, $\vec{x}\sigma$, $P\xi$, $P\theta$, $\vec{y}\xi$, $\vec{y}\theta$, $\vec{R} = [\![\vec{v}]\!]_{\eta,\sigma}$, $\xi' = \xi^{\vec{R}}_{\vec{z}}$ and $\theta' = \theta^{\vec{v}\sigma\, c\vec{x}\sigma}_{\vec{z}\ z}$, and assume that $\vdash_f^<$ is computable, $\eta, \sigma \models \Gamma_c$ and $\eta\xi', \sigma\theta' \models P : T, \vec{y} : \vec{U}$. We must prove that $y_i\theta\vec{x}\sigma\vec{t}\sigma\theta \in [\![P\vec{z}z]\!]_{\xi',\theta'}$.

We have $y_i\theta \in [\![U_i]\!]_{\xi',\theta'}$, $U_i = (\vec{x} : \vec{T})(\vec{x}' : \vec{T}')P\vec{v}(c\vec{q}\vec{x})$ and $x_j\sigma \in [\![T_j]\!]_{\eta,\sigma} = [\![T_j]\!]_{\eta\xi',\sigma\theta'}$. We prove that $t'_j\sigma\theta \in [\![T'_j]\!]_{\eta\xi',\sigma\theta'}$. If $T'_j = T_j$ then $t'_j\sigma\theta = x_j\sigma$ and we are done. Otherwise, $T_j = (\vec{\alpha} : \vec{W})C\vec{q}\vec{w}$, $T'_j = (\vec{\alpha} : \vec{W})P\vec{w}(x_j\vec{\alpha})$ and $t'_j = [\vec{\alpha} : \vec{W}]f\vec{q}\vec{w}(x_j\vec{\alpha})P\vec{y}$. Let $\vec{\alpha}\zeta$ and $\vec{\alpha}\gamma$ such that $\eta\xi'\zeta, \sigma\theta'\gamma \models \vec{\alpha} : \vec{W}$. Let $t = x_j\sigma\vec{\alpha}\gamma$. We must prove that $v = f\vec{q}\sigma\vec{w}\sigma\gamma tP\theta\vec{y}\theta \in S = [\![P\vec{w}(x_j\vec{\alpha})]\!]_{\eta\xi'\zeta,\sigma\theta'\gamma}$. Since v is neutral, it suffices to prove that $\to(v) \subseteq S$.

By (R1), we have $\vec{q}\sigma tP\theta\vec{y}\theta \in \mathcal{SN}$. Since $\vdash_f^<$ is computable and $\vec{w}$ is a subterm of τ_f, by (R1), we also have $\vec{w}\sigma\gamma \in \mathcal{SN}$. Thus, we can proceed by induction on $\vec{q}\sigma\vec{w}\sigma\gamma tP\theta\vec{y}\theta \in \mathcal{SN}$ with $\to$ as well-founded ordering. In the case of a reduction in $\vec{q}\sigma\vec{w}\sigma\gamma tP\theta\vec{y}\theta$, we conclude by induction hypothesis. Assume now that we have a head-reduct v'. By definition of recursors, v' is also a head-reduct of $v_0 = f(\vec{q}\sigma)^*(\vec{w}\sigma\gamma)^*tP\theta\vec{y}\theta$ where $(\vec{q}\sigma)^*(\vec{w}\sigma\gamma)^*$ are the normal forms of $\vec{q}\sigma\vec{w}\sigma\gamma$. If $v_0 \in S$ then, by (R2), $v' \in S$. So, let us prove that $v_0 \in S$.

By candidate substitution (Lemma 40 in [6]), $S = [\![P\vec{z}z]\!]_{\xi^{\vec{S}}_{\vec{z}},\theta^{\vec{w}\sigma\gamma\, t}_{\vec{z}\ z}}$ with $\vec{S} = [\![\vec{w}]\!]_{\eta\xi'\zeta,\sigma\theta'\gamma} = [\![\vec{w}]\!]_{\eta\xi\zeta,\sigma\theta\gamma}$ for $\mathrm{FV}(\vec{w}) \subseteq \{\vec{q}, P, \vec{x}, \vec{\alpha}\}$. Since $x_j\sigma \in [\![T_j]\!]_{\eta\xi',\sigma\theta'}$ and $\eta\xi'\zeta, \sigma\theta'\gamma \models \vec{\alpha} : \vec{W}$, $t \in [\![C\vec{q}\vec{w}]\!]_{\eta\xi'\zeta,\sigma\theta'\gamma} = I_C(\vec{q}\sigma\vec{w}\sigma\gamma, \vec{q}\xi\vec{S})$. Since $\eta\xi', \sigma\theta' \models P : T, \vec{y} : \vec{U}$ and $\mathrm{FV}(T\vec{U}) \subseteq \{\vec{q}, P\}$, we have $\eta\xi, \sigma\theta \models P : T, \vec{y} : \vec{U}$ and $\eta\xi^{\vec{S}}_{\vec{z}}, \sigma\theta^{\vec{w}\sigma\gamma\, t}_{\vec{z}\ z} \models P : T, \vec{y} : \vec{U}$. Therefore, $v_0 \in S$.

The proof is about the same for strong recursors. □

7. Application to CIC

It follows that CAC subsumes CIC almost completely. However, Theorem 6.1 cannot be applied to CIC directly since CIC and CAC do not have the same syntax and the same typing rules. So, we define a sub-system of CIC, called CIC$^-$, whose terms can be translated into a CAC satisfying the conditions of Theorem 6.1.

The ι-reduction of CIC introduces many β-redexes and the recursive calls on $Elim$ are made on bound variables which are later instantiated by structurally smaller terms. Instead, we consider the relation $\to_{\beta\iota'}$ where one step of $\to_{\iota'}$ corresponds to a ι-reduction followed by as many β-reductions as necessary for erasing the β-redexes introduced by the ι-reduction. This is this reduction relation which is actually implemented in the Coq system [8]. Moreover, we conjecture that the strong normalization of $\to_{\beta\iota'}$ implies the strong normalization of $\to_{\beta\iota}$.

Definition 7.1. (ι'-reduction)
The *ι'-reduction* is the reduction relation defined by the rule:

$$Elim(I,Q,\vec{x},Constr(i,I')\,\vec{z})\{\vec{f}\} \;\to_{\iota'}\; \Delta'[I,X,C_i,f_i,Q,\vec{f},\vec{z}]$$

where $I = Ind(X:A)\{\vec{C}\}$ and $\Delta'[I,X,C,f,Q,\vec{f},\vec{z}]$ is defined as follows:

- $\Delta'[I,X,X\vec{m},f,Q,\vec{f},\emptyset] = f$
- $\Delta'[I,X,(z:B)D,f,Q,\vec{f},z\vec{z}] = \Delta'[I,X,D,fz,Q,\vec{z}]$ if $X \notin \mathrm{FV}(B)$
- $\Delta'[I,X,(z:B)D,f,Q,\vec{f},z\vec{z}] = \Delta'[I,X,D,fz\,[\vec{y}:\vec{D}]Elim(I,Q,\vec{q},z\vec{y}),Q,\vec{z}]$ if $B = (\vec{y}:\vec{D})X\vec{q}$

We now define the sub-system of CIC (see Figure 3) that we are going to consider:

Definition 7.2. (CIC$^-$)

- We exclude any use of the sort $\triangle$ in order to stay in the Calculus of Constructions.
- In the rule (conv), instead of requiring $T \leftrightarrow^*_{\beta\eta\iota} T'$, we require $T \leftrightarrow^*_{\beta\iota'} T'$ which is equivalent to $T \downarrow_{\beta\iota'} T'$ since $\to_{\beta\iota'}$ is confluent (orthogonal CRS [25]).
- In the rule (Ind), we require I to be in normal form w.r.t. $\to_{\beta\iota'}$ (set $\mathcal{NF}$) and to be typable in the empty environment since, in CAC, the types of symbols must be typable in the empty environment. This is not a real restriction since any type $I = Ind(X:A)\{\vec{C}\}$ typable in an environment $\Gamma = \vec{y}:\vec{U}$ can be replaced by a type $I' = Ind(X':A')\{\vec{C'}\}$ typable in the empty environment. It suffices to take $A' = (\vec{y}:\vec{U})A$, $C'_i = (\vec{y}:\vec{U})C_i\{X \mapsto X'\vec{y}\}$ and to replace I by $I'\vec{y}$ and $Constr(i,I)$ by $Constr(i,I')\vec{y}$. Furthermore, we adapt the definition of *small* constructor type accordingly. A constructor type C of an inductive type $I = Ind(X:A)\{\vec{C}\}$ with $A = (\vec{x}:\vec{A})\star$ is *small* if it is of the form $(\vec{x'}:\vec{A'})(\vec{z}:\vec{B})X\vec{m}$ with $\vec{x'}:\vec{A'}$ a sub-sequence of $\vec{x}:\vec{A}$ and $\{\vec{z}\} \cap \mathcal{X}^\Box = \emptyset$.
- In the rule ($\star$-Elim), we require Q to be typable in the empty environment, and add explicit typing judgments for T_i and I. Again, it is not a real restriction since we can always replace an environment by additional abstractions.
- In the rule ($\Box$-Elim), instead of requiring $\vdash Q : (\vec{x}:\vec{A})I\vec{x} \Rightarrow \Box$, which is not possible in CC, we require Q to be of the form $[\vec{x}:\vec{A}][y:I\vec{x}]K$ with $\vec{x}:\vec{A}, y:I\vec{x} \vdash K : \Box$ (this just requires some η-expansions) and f_i to be of type $T_i = \Delta'\{I,X,C_i,\vec{x}y,K,Constr(i,I)\}$ where $\Delta'\{I,X,C,\vec{x}y,K,c\}$ is defined as follows:
  - $\Delta'\{I,X,X\vec{m},\vec{x}y,K,c\} = K\{\vec{x}\mapsto\vec{m},y\mapsto c\}$,
  - $\Delta'\{I,X,(z:B)D,\vec{x}y,K,c\} =$
    $(z:B\{X\mapsto I\})((\vec{y}:\vec{D})K\{\vec{x}\mapsto\vec{q},y\mapsto z\vec{y}\}) \Rightarrow \Delta'\{I,X,D,\vec{x}y,K,cz\}$ if $B = (\vec{y}:\vec{D})X\vec{q}$.

  Moreover, we require Q to be in normal form and T_i to be typable. We also take $\Gamma \vdash Elim(I,Q,\vec{a},c)\{\vec{f}\} : K\{\vec{x}\mapsto\vec{a},y\mapsto c\}$ instead of $\Gamma \vdash Elim(I,Q,\vec{a},c)\{\vec{f}\} : Q\vec{a}c$. Finally, we require I to be safe (see Definition 1.2): if $A = (\vec{x}:\vec{A})\star$ and $C_i = (\vec{z}:\vec{B})X\vec{m}$ then:
  - for all $x_i \in \mathcal{X}^\Box$, $m_i \in \mathcal{X}^\Box$,
  - for all $x_i, x_j \in \mathcal{X}^\Box$ with $i \neq j$, $m_i \neq m_j$.

We now show that CIC$^-$ can be translated into a CAC satisfying the conditions of Theorem 6.1.

Definition 7.3. (Translation)
We define $\langle t\rangle$ on well-typed terms, by induction on $\Gamma \vdash t : T$:

Figure 3. Typing rules of CIC $^-$

$$\text{(Ind)}\quad \frac{A=(\vec{x}:\vec{A})\star \quad \vdash A:\Box \quad \forall i,\ X:A\vdash C_i:\star \qquad I=Ind(X:A)\{\vec{C}\}\in\mathcal{NF} \text{ is strictly positive}}{\vdash I:A}$$

$$\text{(Constr)}\quad \frac{I=Ind(X:A)\{\vec{C}\}\quad \Gamma\vdash I:T}{\Gamma\vdash Constr(i,I):C_i\{X\mapsto I\}}$$

$$\text{($\star$-Elim)}\quad \frac{\begin{array}{c}A=(\vec{x}:\vec{A})\star \quad I=Ind(X:A)\{\vec{C}\}\quad \Gamma\vdash I:T\quad \vdash Q:(\vec{x}:\vec{A})I\vec{x}\Rightarrow\star\\ T_i=\Delta\{I,X,C_i,Q,Constr(i,I)\}\quad \vdash T_i:\star\\ \forall j,\ \Gamma\vdash a_j:A_j\{\vec{x}\mapsto\vec{a}\}\quad \Gamma\vdash c:I\vec{a}\quad \forall i,\ \Gamma\vdash f_i:T_i\end{array}}{\Gamma\vdash Elim(I,Q,\vec{a},c)\{\vec{f}\}:Q\vec{a}c}$$

$$\text{($\Box$-Elim)}\quad \frac{\begin{array}{c}A=(\vec{x}:\vec{A})\star \quad I=Ind(X:A)\{\vec{C}\}\text{ is small and safe}\\ Q=[\vec{x}:\vec{A}][y:I\vec{x}]K\in\mathcal{NF}\quad \vec{x}:\vec{A},y:I\vec{x}\vdash K:\Box\\ T_i=\Delta'\{I,X,C_i,\vec{x}y,K,Constr(i,I)\}\quad \vdash T_i:\Box\\ \forall j,\ \Gamma\vdash a_j:A_j\{\vec{x}\mapsto\vec{a}\}\quad \Gamma\vdash c:I\vec{a}\quad \forall i,\ \Gamma\vdash f_i:T_i\end{array}}{\Gamma\vdash Elim(I,Q,\vec{a},c)\{\vec{f}\}:K\{\vec{x}\mapsto\vec{a},y\mapsto c\}}$$

$$\text{(Conv)}\quad \frac{\Gamma\vdash t:T\quad T\leftrightarrow^*_{\beta\iota'}T'\quad \Gamma\vdash T':s}{\Gamma\vdash t:T'}$$

- If $I=Ind(X:A)\{\vec{C}\}$ then $\langle I\rangle=Ind_I$ where Ind_I is a symbol of type $\langle A\rangle$.
- $\langle Constr(i,I)\rangle=Constr_i^I$ where $Constr_i^I$ is a symbol of type $\langle C_i\{X\mapsto I\}\rangle$.
- If Q is not of the form $[\vec{x}:\vec{A}][y:I\vec{x}](\vec{y}:\vec{U})\star$ then $\langle Elim(I,Q,\vec{a},c)\{\vec{f}\}\rangle=WElim_I\langle Q\rangle\langle\vec{a}\rangle\langle c\rangle\langle\vec{f}\rangle$ where $WElim_I$ is a symbol of type $(Q:(\vec{x}:\langle\vec{A}\rangle)\langle I\rangle\vec{x}\Rightarrow\star)(\vec{x}:\langle\vec{A}\rangle)(y:\langle I\rangle\vec{x})(\vec{f}:\langle\vec{T}\rangle)\langle Q\rangle\vec{x}y$.
- If $Q=[\vec{x}:\vec{A}][y:I\vec{x}]K$ with $K=(\vec{y}:\vec{U})\star$ then $\langle Elim(I,Q,\vec{a},c)\{\vec{f}\}\rangle=SElim_I^Q\langle\vec{a}\rangle\langle c\rangle\langle\vec{f}\rangle$ where $SElim_I^Q$ is a symbol of type $(\vec{x}:\langle\vec{A}\rangle)(y:\langle I\rangle\vec{x})(\vec{f}:\langle\vec{T}\rangle)\langle K\rangle$.
- The translation of the other terms is defined recursively: $\langle uv\rangle=\langle u\rangle\langle v\rangle,\ldots$

Let Υ be the CAC whose symbols are $Ind_I, Constr_i^I, WElim_I$ and $SElim_I^Q$, and whose rules are:

$$\begin{array}{rcl}WElim_I\ Q\ \vec{x}\ (Constr_i^I\ \vec{z})\ \vec{f} & \to & \Delta'_W[I,X,C_i,f_i,Q,\vec{f},\vec{z}]\\ SElim_I^Q\ \vec{x}\ (Constr_i^I\ \vec{z})\ \vec{f} & \to & \Delta'_S[I,X,C_i,f_i,Q,\vec{f},\vec{z}]\end{array}$$

where $\Delta'_W[I,X,C,f,Q,\vec{f},\vec{z}]$ and $\Delta'_S[I,X,C,f,Q,\vec{f},\vec{z}]$ are defined as follows:

– $\Delta'_W[I,X,X\vec{m},f,Q,\vec{f},\vec{z}]=\Delta'_S[I,X,X\vec{m},f,Q,\vec{f},\vec{z}]=f$,

– $\Delta'_S[I, X, (z : B)D, f, Q, \vec{f}, z\vec{z}] = \Delta'_S[I, X, D, f\ z, Q, \vec{f}, \vec{z}]$ and
$\Delta'_W[I, X, (z : B)D, f, Q, \vec{f}, z\vec{z}] = \Delta'_W[I, X, D, f\ z, Q, \vec{f}, \vec{z}]$ if $X \notin \mathrm{FV}(B)$
– $\Delta'_S[I, X, (z : B)D, f, Q, \vec{f}, z\vec{z}] = \Delta'_S[I, X, D, f\ z\ [\vec{y} : \vec{D}]SElim_I^Q\ \vec{f}\vec{q}(z\vec{y}), Q, \vec{f}, \vec{z}]$ and
$\Delta'_W[I, X, (z : B)D, f, Q, \vec{f}, z\vec{z}] = \Delta'_W[I, X, D, f\ z\ [\vec{y} : \vec{D}]WElim_I Q\vec{f}\vec{q}(z\vec{y}), Q, \vec{f}, \vec{z}]$
if $B = (\vec{y} : \vec{D})X\vec{q}$

Let $\vdash_\Upsilon$ be the typing relation of Υ.

Theorem 7.1. The relation $\rightarrow_{\beta\iota'}$ in CIC$^-$ preserves typing and is strongly normalizing.

Proof:
First, one can easily check that the translation preserves typing and reductions:
– If $\Gamma \vdash t : T$ then $\langle\Gamma\rangle \vdash_\Upsilon \langle t\rangle : \langle T\rangle$.
– If $\Gamma \vdash t : T$ and $t \rightarrow_{\beta\iota'} t'$ then $\langle t\rangle \rightarrow \langle t'\rangle$.

Thus, we are left to prove that Υ satisfies the conditions of Theorem 6.1. The symbols $WElim_I$ and $SElim_I^Q$ are the canonical recursors of Ind_I w.r.t. the constructors $Constr_i^I$ (see Definition 6.2). Hence, subject reduction follows from Lemma 6.4, and the fact that $\mathcal{R}ec(Ind_I) = \{WElim_I, SElim_I^Q\}$ is admissible follows from Lemma 6.5 and Lemma 6.6. □

8. Non-strictly positive types

We are going to see that the use of elimination-based interpretations allows us to have functions defined by recursion on non-strictly positive types, while CIC has always been restricted to strictly positive types. An interesting example is given by Abel's formalization of first-order terms with continuations as an inductive type $trm : \star$ with the constructors [1]:

$$\begin{aligned} var &: nat \Rightarrow trm \\ fun &: nat \Rightarrow (list\ trm) \Rightarrow trm \\ mu &: \neg\neg trm \Rightarrow trm \end{aligned}$$

where $list : \star \Rightarrow \star$ is the type of polymorphic lists, $\neg X$ is an abbreviation for $X \Rightarrow \bot$ (in the next section, we will prove that $\neg$ can be defined as a function), and $\bot : \star$ is the empty type. Its recursor $rec : (A : \star)(y_1 : nat \Rightarrow A)\ (y_2 : nat \Rightarrow list\ trm \Rightarrow listA \Rightarrow A)(y_3 : \neg\neg trm \Rightarrow \neg\neg A \Rightarrow A)(z : trm)A$ can be defined by the rules:

$$\begin{aligned} rec\ A\ y_1\ y_2\ y_3\ (var\ n) &\rightarrow y_1\ n \\ rec\ A\ y_1\ y_2\ y_3\ (fun\ n\ l) &\rightarrow y_2\ n\ l\ (map\ trm\ A\ (rec\ A\ y_1\ y_2\ y_3)\ l) \\ rec\ A\ y_1\ y_2\ y_3\ (mu\ f) &\rightarrow y_3\ f\ [x : \neg A](f\ [y : trm](x\ (rec\ A\ y_1\ y_2\ y_3\ y))) \end{aligned}$$

where $map : (A : \star)(B : \star)(A \Rightarrow B) \Rightarrow list\ A \Rightarrow list\ B$ is defined by the rules:

$$\begin{aligned} map\ A\ B\ f\ (nil\ A') &\rightarrow (nil\ B) \\ map\ A\ B\ f\ (cons\ A'\ x\ l) &\rightarrow cons\ B\ (f\ x)\ (map\ A\ B\ f\ l) \\ map\ A\ B\ f\ (app\ A'\ l\ l') &\rightarrow app\ B\ (map\ A\ B\ f\ l)\ (map\ A\ B\ f\ l') \end{aligned}$$

We now check that rec is an admissible recursor. Completeness w.r.t. accessibility is easy. For the head-computability, we only detail the case of mu. Let $f\sigma$, $t = mu\ f\sigma$, $A\xi$, $A\theta$ and $\vec{y}\theta$ such that $\emptyset, \sigma \models \Gamma_{mu}$ and $\xi, \sigma\theta_z^t \models \Gamma = A : \star, \vec{y} : \vec{U}$ where U_i is the type of y_i. Let $b = recA\theta\vec{y}\theta$, $c = [y : trm](x(by))$ and $a = [x : \neg A\theta](f\sigma c)$. We must prove that $y_3\theta f\sigma a \in [\![A]\!]_{\xi,\sigma\theta_z^t} = A\xi$.

Since $\xi, \sigma\theta_z^t \models \Gamma$, $y_3\theta \in [\![\neg\neg trm \Rightarrow \neg\neg A \Rightarrow A]\!]_{\xi,\theta}$. Since $\emptyset, \sigma \models \Gamma_{mu}$, $f\sigma \in [\![\neg\neg trm]\!]$. Thus, we are left to prove that $a \in [\![\neg\neg A]\!]_{\xi,\theta}$, that is, $f\sigma c\gamma \in I_\perp$ for all $x\gamma \in [\![\neg A]\!]_{\xi,\theta}$. Since $f\sigma \in [\![\neg\neg trm]\!]$, it suffices to prove that $c\gamma \in [\![\neg trm]\!]$, that is, $x\gamma(by\gamma) \in I_\perp$ for all $y\gamma \in I_{trm}$. This follows from the facts that $x\gamma \in [\![\neg A]\!]_{\xi,\theta}$ and $by\gamma \in A\xi$ since $y\gamma \in I_{trm}$.

A general proof could certainly be given by using a general formalization of inductive types like in [19] for instance.

9. Inductive-recursive types

In this section, we define new positivity conditions for dealing with *inductive-recursive type definitions* [13]. An inductive-recursive type C has constructors whose arguments have a type Ft with F defined by recursion on $t : C$, that is, a predicate F and its domain C are defined at the same time.

A simple example is the type $dlist : (A : \star)(\# : A \Rightarrow A \Rightarrow \star)\star$ of lists made of distinct elements thanks to the predicate $fresh : (A : \star)(\# : A \Rightarrow A \Rightarrow \star)A \Rightarrow (dlist\ A\ \#) \Rightarrow \star$ parametrized by a function $\#$ to test whether two elements are distinct. The constructors of $dlist$ are:

$$\begin{aligned} nil &: (A : \star)(\# : A \Rightarrow A \Rightarrow \star)(dlist\ A\ \#) \\ cons &: (A : \star)(\# : A \Rightarrow A \Rightarrow \star)(x : A)(l : dlist\ A\ \#)(fresh\ A\ \#\ x\ l) \Rightarrow (dlist\ A\ \#) \end{aligned}$$

and the rules defining $fresh$ are:

$$\begin{aligned} fresh\ A\ \#\ x\ (nil\ A') &\rightarrow \top \\ fresh\ A\ \#\ x\ (cons\ A'\ y\ l\ h) &\rightarrow x\#y \wedge fresh\ A\ \#\ x\ l \end{aligned}$$

where $\top$ is the proposition always true and $\wedge$ the connector "and". Other examples are given by Martin-Löf's definition of the first universe *à la* Tarski [13] or by Pollack's formalization of record types with manifest fields [27].

For allowing defined predicate symbols in constructor types, we must extend the notion of positive and negative positions by taking into account the arguments in which a defined predicate symbol is monotone or anti-monotone. We must also make sure that defined predicate symbols are indeed monotone and anti-monotone in the arguments declared to have this property.

Definition 9.1. (Positive/negative positions - New definition)
Assume that every predicate symbol $f : (\vec{x} : \vec{T})U$ with U not a product is equipped with a set $\mathrm{Mon}^+(f) \subseteq A_f^\Box = \{i \leq |\vec{x}| \mid x_i \in \mathcal{X}^\Box\}$ of *monotone arguments* and a set $\mathrm{Mon}^-(f) \subseteq A_f^\Box$ of *anti-monotone arguments*. Definition 4.4 is modified as follows:

- $\mathrm{Pos}^\delta(f\vec{t}) = \{1^{|\vec{t}|} \mid \delta = +\} \cup \bigcup\{1^{|\vec{t}|-i}2.\mathrm{Pos}^{\epsilon\delta}(t_i) \mid \epsilon \in \{-,+\},\ i \in \mathrm{Mon}^\epsilon(f)\}$,
- $\mathrm{Pos}^\delta(tu) = 1.\mathrm{Pos}^\delta(t)$ if t is not of the form $f\vec{t}$.

For instance, in the positive type trm of Section 8, instead of considering $\neg A$ as an abbreviation, one can consider $\neg$ as a predicate symbol defined by the rule $\neg A \to A \Rightarrow \bot$ with $\mathrm{Mon}^-(\neg) = \{1\}$. Then, one easily check that A occurs negatively in $A \Rightarrow \bot$, and hence that trm occurs positively in $\neg\neg trm$ since $\mathrm{Pos}^+(\neg\neg trm) = \{1\} \cup 2.\mathrm{Pos}^-(\neg trm) = \{1\} \cup 2.2.\mathrm{Pos}^+(trm) = \{1, 2.2\}$.

Definition 9.2. (Positivity conditions - New definition)
Definition 5.2 is modified as follows. A pre-recursor $f : (\vec{z} : \vec{V})(z : C\vec{z})W$ is a *recursor* if:

– every $F \simeq C$ occurs only positively in W,
– if $i \in \mathrm{Mon}^\delta(C)$ then $\mathrm{Pos}(z_i, W) \subseteq \mathrm{Pos}^\delta(W)$.

Moreover, we assume that, for every rule $F\vec{l} \to r \in \mathcal{R}$ with $F \in \mathcal{F}^\Box$:

– for all $i \in \mathrm{Mon}^\epsilon(F)$, $l_i \in \mathcal{X}^\Box$ and $\mathrm{Pos}(l_i, r) \subseteq \mathrm{Pos}^\epsilon(r)$.

Now, we must reflect these monotony properties in the interpretations. Then, Theorem 6.1 is still valid if we prove that the interpretations for constant and defined predicate symbols have all the monotony properties.

Definition 9.3. (Monotone interpretation)
Let $\vec{S} \leq_i \vec{S'}$ iff $S_i \leq S'_i$ and, for all $j \neq i$, $S_j = S'_j$. Let F be a predicate symbol. An interpretation $I \in \mathcal{R}_{\tau_F}$ is *monotone* (resp. *anti-monotone*) *in its i-th argument* if $I(\vec{t}, \vec{S}) \leq I(\vec{t}, \vec{S'})$ whenever $\vec{S} \leq_i \vec{S'}$ (resp. $\vec{S} \geq_i \vec{S'}$). An interpretation $I \in \mathcal{R}_{\tau_F}$ is *monotone* if it is monotone in every $i \in \mathrm{Mon}^+(F)$ and anti-monotone in every $i \in \mathrm{Mon}^-(F)$. Let $\mathcal{R}^m_{\tau_F}$ be the set of monotone interpretations of $\mathcal{R}_{\tau_F}$.

One can easily check that $\mathcal{R}^m_{\tau_F}$ is a complete lattice too. For proving that interpretations for predicate symbols are monotone, we need to prove Lemma 5.2 again, and to prove a similar lemma on candidate assignments.

Lemma 9.1. If $I \leq_f I'$, $\mathrm{Pos}(f, t) \subseteq \mathrm{Pos}^\delta(t)$, $\Gamma \vdash t : T$ and $\xi \models \Gamma$ then $[\![t]\!]^I_{\xi,\theta} \leq^\delta [\![t]\!]^{I'}_{\xi,\theta}$.

Proof:
We only have to check the case $t = g\vec{t}$. Let $R = [\![g\vec{t}]\!]^I_{\xi,\theta}$ and $R' = [\![g\vec{t}]\!]^{I'}_{\xi,\theta}$. $R = I_g(\vec{t\theta}, \vec{S})$ with $\vec{S} = [\![\vec{t}]\!]^I_{\xi,\theta}$. $R' = I_g(\vec{t\theta}, \vec{S'})$ with $\vec{S'} = [\![\vec{t}]\!]^{I'}_{\xi,\theta}$. Let $i \leq n = |\vec{t}|$. If $\mathrm{Pos}(f, t_i) = \emptyset$ then $S_i = S'_i$. Otherwise, there is ϵ_i such that $i \in \mathrm{Mon}^{\epsilon_i}(f)$ and $\mathrm{Pos}(f, t_i) \subseteq \mathrm{Pos}^{\epsilon_i\delta}(t_i)$. Thus, by induction hypothesis, $S_i \leq^{\epsilon_i\delta} S'_i$. Let $S^j_i = S_i$ if $i > j$, and $S^j_i = S'_i$ otherwise. $\vec{S}^0 = \vec{S}$, $\vec{S}^n = \vec{S'}$ and, for all $j \leq n$, $\vec{S}^{j-1} \leq^{\epsilon_j\delta}_j \vec{S}^j$. Since I_g is monotone, for all $j \leq n$, $I_g(\vec{t\theta}, \vec{S}^{j-1}) \leq^{\epsilon_j^2\delta} I_g(\vec{t\theta}, \vec{S}^j)$, that is, $I_g(\vec{t\theta}, \vec{S}^{j-1}) \leq^\delta I_g(\vec{t\theta}, \vec{S}^j)$ since $\epsilon_j^2 = +$. Thus, $R = I_g(\vec{S}) \leq^\delta I_g(\vec{S'})$. Now, if $g \neq f$ then $I_g = I'_g$ and $R \leq^\delta R'$. If $g = f$ then $\delta = +$ and $R \leq R'$ since $I_f \leq I'_f$. □

Lemma 9.2. Let $\xi \leq_x \xi'$ iff $x\xi \leq x\xi'$ and, for all $y \neq x$, $y\xi = y\xi'$. If I is monotone, $\xi \leq_x \xi'$, $x \in \mathrm{Pos}^\delta(t)$, $\Gamma \vdash t : T$ and $\xi, \xi' \models \Gamma$ then $[\![t]\!]^I_{\xi,\theta} \leq^\delta [\![t]\!]^I_{\xi',\theta}$.

Proof:
By induction on t. The proof is very similar to the previous lemma. We only detail the following two cases:

- $[\![x]\!]^I_{\xi,\theta} = x\xi \leq x\xi' = [\![x]\!]^I_{\xi,\theta}$ and $\delta = +$ necessarily.

- Let $R = [\![g\vec{t}]\!]^I_{\xi,\theta}$ and $R' = [\![g\vec{t}]\!]^I_{\xi',\theta}$. $R = I_g(\vec{t}\theta, \vec{S})$ with $\vec{S} = [\![\vec{t}]\!]^I_{\xi,\theta}$. $R' = I_g(\vec{t}\theta, \vec{S}')$ with $\vec{S}' = [\![\vec{t}]\!]^I_{\xi',\theta}$. Let $i \leq n = |\vec{t}|$. If $\mathrm{Pos}(f, t_i) = \emptyset$ then $S_i = S'_i$. Otherwise, there is ϵ_i such that $i \in \mathrm{Mon}^{\epsilon_i}(f)$ and $\mathrm{Pos}(f, t_i) \subseteq \mathrm{Pos}^{\epsilon_i\delta}(t_i)$. Thus, by induction hypothesis, $S_i \leq^{\epsilon_i\delta} S'_i$. Let $S^j_i = S_i$ if $i > j$, and $S^j_i = S'_i$ otherwise. $\vec{S}^0 = \vec{S}$, $\vec{S}^n = \vec{S}'$ and, for all $j \leq n$, $\vec{S}^{j-1} \leq^{\epsilon_j\delta}_j \vec{S}^j$. Since I_g is monotone, for all $j \leq n$, $I_g(\vec{t}\theta, \vec{S}^{j-1}) \leq^{\epsilon_j^2\delta} I_g(\vec{t}\theta, \vec{S}^j)$, that is, $I_g(\vec{t}\theta, \vec{S}^{j-1}) \leq^{\delta} I_g(\vec{t}\theta, \vec{S}^j)$ since $\epsilon_j^2 = +$. Thus, $R \leq^{\delta} R'$.

□

Lemma 9.3. The interpretations for predicate symbols are monotone.

Proof:
We first prove it for constant predicate symbols. Assuming that I is monotone, we must prove that φ^I_C is monotone. Let $i \in \mathrm{Mon}^{\delta}(C)$ and $\vec{S} \leq^{\delta}_i \vec{S}'$. We must prove that $R = \varphi^I_C(\vec{t}, \vec{S}) \subseteq R' = \varphi^I_C(\vec{t}, \vec{S}')$. If some t_i has no normal form then $R = R' = \mathcal{SN}$. Assume now that every t_i has a normal form t_i^*. Let $t \in R$, $f \in \mathcal{R}ec(C)$ of type $(\vec{z} : \vec{V})(z : C\vec{z})(\vec{y} : \vec{U})V$, $\vec{y}\xi$ and $\vec{y}\theta$ such that $\xi^{\vec{S}'}_{\vec{z}}, \theta^{\vec{t}\,t}_{\vec{z}z} \models_I \vec{y} : \vec{U}$. We must prove that $f\vec{t}^*t\vec{y}\theta \in [\![V]\!]^I_{\xi^{\vec{S}'}_{\vec{z}}, \theta^{\vec{t}\,t}_{\vec{z}z}}$. To this end, it is sufficient to prove that $[\![\vec{U}]\!]^I_{\xi^{\vec{S}'}_{\vec{z}}, \theta^{\vec{t}\,t}_{\vec{z}z}} \subseteq [\![\vec{U}]\!]^I_{\xi^{\vec{S}}_{\vec{z}}, \theta^{\vec{t}\,t}_{\vec{z}z}}$ and that $[\![V]\!]^I_{\xi^{\vec{S}}_{\vec{z}}, \theta^{\vec{t}\,t}_{\vec{z}z}} \subseteq [\![V]\!]^I_{\xi^{\vec{S}'}_{\vec{z}}, \theta^{\vec{t}\,t}_{\vec{z}z}}$, which is the case by Lemma 9.2 since $\mathrm{Pos}(z_i, W) \subseteq \mathrm{Pos}^+(W)$ by assumption.

We now prove that the interpretation for defined predicate symbols is monotone. Let F be a defined predicate symbol. Let $i \in \mathrm{Mon}^{\delta}(F)$ and $\vec{S} \leq^{\delta}_i \vec{S}'$. We must prove that $R = I_F(\vec{t}, \vec{S}) \subseteq R' = I_F(\vec{t}, \vec{S}')$. Assume that every t_i has a normal form t_i^* and that $\vec{t}^* = \vec{l}\sigma$ for some rule $F\vec{l} \to r \in \mathcal{R}$. If this is not the case then $R = R' = \mathcal{SN}$. So, $R = [\![r]\!]^I_{\xi,\sigma}$ with $x\xi = S_{\kappa_x}$, and $R' = [\![r]\!]^I_{\xi',\sigma}$ with $x\xi' = S'_{\kappa_x}$. If, for all $x \in \mathrm{FV}^{\Box}(r)$, $\kappa_x \neq i$, then $\xi = \xi'$ and $R = R'$. Otherwise, $i = \kappa_x$ for some x, and $\xi \leq^{\delta}_x \xi'$. By Lemma 9.2, $R \subseteq^{\delta^2} R'$ since $\mathrm{Pos}(x, r) \subseteq \mathrm{Pos}^{\delta}(r)$ by assumption. Thus, $R \subseteq R'$ since $\delta^2 = +$. □

10. Conclusion

By using an elimination-based interpretation for some inductive types, we proved that the Calculus of Algebraic Constructions subsumes the Calculus of Inductive Constructions almost completely. We define general conditions on recursors for preserving strong normalization and show that these conditions are satisfied by a large class of recursors for strictly positive types and by some non-strictly positive types too. Finally, we give general positivity conditions for dealing with inductive-recursive types.

Acknowledgments. I would like to thank very much Christine Paulin, Ralph Matthes, Jean-Pierre Jouannaud, Daria Walukiewicz-Chrząszcz, Gilles Dowek and the anonymous referees for their useful comments on previous versions of this paper. Part of this work was performed during my stay at Cambridge (UK) in 2002 with Larry Paulson thanks to a grant from the INRIA.

References

[1] Abel, A.: *Termination Checking with Types*, Technical Report 0201, Ludwig Maximilians Universität, München, Germany, 2002.

[2] Barbanera, F., Fernández, M., Geuvers, H.: Modularity of strong normalization and confluence in the algebraic-λ-cube, *Proceedings of the 9th IEEE Symposium on Logic in Computer Science*, 1994.

[3] Barendregt, H.: Lambda Calculi with types, in: *Handbook of logic in computer science* (S. Abramski, D. Gabbay, T. Maibaum, Eds.), vol. 2, Oxford University Press, 1992.

[4] Blanqui, F.: Definitions by rewriting in the Calculus of Constructions (extended abstract), *Proceedings of the 16th IEEE Symposium on Logic in Computer Science*, 2001.

[5] Blanqui, F.: *Théorie des Types et Récriture*, Ph.D. Thesis, Université Paris XI, Orsay, France, 2001, Available in english as "Type Theory and Rewriting".

[6] Blanqui, F.: Definitions by rewriting in the Calculus of Constructions, *Mathematical Structures in Computer Science*, **15**(1), 2005, 37–92.

[7] Breazu-Tannen, V., Gallier, J.: Polymorphic Rewriting Conserves Algebraic Strong Normalization, *Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science 372, 1989.

[8] Coq-Development-Team: *The Coq Proof Assistant Reference Manual - Version 8.0*, INRIA Rocquencourt, France, 2004, `http://coq.inria.fr/`.

[9] Coquand, T.: An Analysis of Girard's Paradox, *Proceedings of the 1st IEEE Symposium on Logic in Computer Science*, 1986.

[10] Coquand, T., Huet, G.: The Calculus of Constructions, *Information and Computation*, **76**(2-3), 1988, 95–120.

[11] Coquand, T., Paulin-Mohring, C.: Inductively defined types, *Proceedings of the International Conference on Computer Logic*, Lecture Notes in Computer Science 417, 1988.

[12] Dershowitz, N., Jouannaud, J.-P.: Rewrite Systems, in: *Handbook of Theoretical Computer Science* (J. van Leeuwen, Ed.), vol. B, chapter 6, North-Holland, 1990.

[13] Dybjer, P.: A general formulation of simultaneous inductive-recursive definitions in type theory, *Journal of Symbolic Logic*, **65**(2), 2000, 525–549.

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

[15] Harper, R., Mitchell, J.: Parametricity and variants of Girard's J operator, *Information Processing Letters*, **70**, 1999, 1–5.

[16] Jouannaud, J.-P., Okada, M.: Executable Higher-Order Algebraic Specification Languages, *Proceedings of the 6th IEEE Symposium on Logic in Computer Science*, 1991.

[17] Jouannaud, J.-P., Okada, M.: Abstract Data Type Systems, *Theoretical Computer Science*, **173**(2), 1997, 349–391.

[18] Klop, J. W., van Oostrom, V., van Raamsdonk, F.: Combinatory reduction systems: introduction and survey, *Theoretical Computer Science*, **121**, 1993, 279–308.

[19] Matthes, R.: *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*, Ph.D. Thesis, Ludwig Maximilians Universität, München, Germany, 1998.

[20] McBride, C.: *Dependently typed functional programs and their proofs*, Ph.D. Thesis, University of Edinburgh, United Kingdom, 1999.

[21] Mendler, N. P.: *Inductive Definition in Type Theory*, Ph.D. Thesis, Cornell University, United States, 1987.

[22] Müller, F.: Confluence of the lambda calculus with left-linear algebraic rewriting, *Information Processing Letters*, **41**(6), 1992, 293–299.

[23] Nederpelt, R.: *Strong normalization in a typed lambda calculus with lambda structured types*, Ph.D. Thesis, Technische Universiteit Eindhoven, The Netherlands, 1973.

[24] Okada, M.: Strong Normalizability for the Combined System of the Typed Lambda Calculus and an Arbitrary Convergent Term Rewrite System, *Proceedings of the 1989 International Symposium on Symbolic and Algebraic Computation*, ACM Press.

[25] van Oostrom, V.: *Confluence for Abstract and Higher-Order Rewriting*, Ph.D. Thesis, Vrije Universiteit Amsterdam, The Netherlands, 1994.

[26] Paulin-Mohring, C.: Personal communication, 2001.

[27] Pollack, R.: Dependently typed records in type theory, *Formal Aspects of Computing*, **13**(3-5), 2002, 341–363.

[28] Rusinowitch, M.: On termination of the direct sum of term-rewriting systems, *Information Processing Letters*, **26**(2), 1987, 65–70.

[29] Stefanova, M.: *Properties of Typing Systems*, Ph.D. Thesis, Katholiecke Universiteit Nijmegen, The Netherlands, 1998.

[30] Toyama, Y.: Counterexamples to termination for the direct sum of term rewriting systems, *Information Processing Letters*, **25**(3), 1987, 141–143.

[31] Walukiewicz-Chrząszcz, D.: Termination of rewriting in the Calculus of Constructions, *Journal of Functional Programming*, **13**(2), 2003, 339–414.

[32] Werner, B.: *Une Théorie des Constructions Inductives*, Ph.D. Thesis, Université Paris VII, France, 1994.