# Codifying guarded definitions with recursive schemes

Eduardo Giménez [*]

LIP-IMAG URA CNRS 1398, ENS Lyon
46, Allée d'Italie 69364 Lyon Cedex 07, France
email : `egimenez@lip.ens-lyon.fr`

**Abstract.** We formalize an extension of the Calculus of Constructions with inductive and coinductive types which allows a more direct description of recursive definitions. The approach we follow is close to the one proposed for Martin-Löf's type theory in [5]. Recursive objects can be defined by fixed-point definitions as in functional programming languages, and a syntactical checking of these definitions avoids the introduction of non-normalizable terms. We show that the conditions for accepting a recursive definition proposed in [5] are not sufficient for the Calculus of Constructions, and we modify them. As a way of justifying our conditions, we develop a general method to codify a fix point definition satisfying them using well-known recursive schemes, like primitive recursion and co-recursion. We also propose different reduction rules from the ones used in [5] in order to obtain a decidable conversion relation for the system.

## 1 Introduction

Using the "Curry-Howard correspondence" it is possible to describe proofs in natural deduction as lambda terms, and thus to consider typed lambda calculus as the basis of a proof system. The extension of this basic calculus with primitive inductive types allows a direct representation of several mathematical notions in the system, just providing the constructors (introduction rules) of these notions.

The introduction of primitive inductive types supposes to extend the system also with some way of expressing recursive definitions, so that we can program and reason with the elements of these types. Since recursive definitions introduce non-canonical terms, the question of normalization is immediately posed : has any closed term with an inductive type (at least) a head-of-constructor form? From a logical point of view, this question is directly bound to the consistency of the proof system. In proof systems based on type theory, consistency means that there is a type which has no inhabitant. If every term can be normalized to a head-of-constructor form this property is ensured, since there can not exist an element in the empty type (the type which has no constructor).

[*] This research was partially supported by ESPRIT Basic Research Action "Types for Proofs and Programs" and by Programme de Recherches Coordonnées and CNRS Groupement de Recherche "Programmation".

A possible way of expressing recursion without introducing non-normalizable terms is to associate an elimination rule to each inductive type. In [12] it is described how to generate such a rule for a large class of types. For example, the elimination rule associated to the type Nat whose constructors are $0 : \mathsf{Nat}$ and $\mathsf{s} : \mathsf{Nat} \rightarrow \mathsf{Nat}$ is a defined constant:

$$\mathsf{Nrec} : (C : \mathsf{Nat} \rightarrow Set)(C\ 0) \rightarrow ((y : \mathsf{Nat})(C\ y) \rightarrow (C\ (\mathsf{s}\ y))) \rightarrow (x : \mathsf{Nat})(C\ x)$$

This constant represents the induction principle for natural numbers. The reduction rules associated to it define a safe recursive scheme for programming (primitive recursion).

$$\begin{array}{lll}(\mathsf{Nrec}\ C\ b\ h\ 0) & = b & : (C\ 0)\\ (\mathsf{Nrec}\ C\ b\ h\ (\mathsf{s}\ n)) & = (h\ n\ (\mathsf{Nrec}\ C\ h\ b\ n)) & : (C\ (\mathsf{s}\ n))\end{array}$$

In this approach, the only way of describing a function on Nat is to codify it using Nrec. This leads to a strong limitation in the expressiveness of the proof system. From a theoretical point of view, a large space of functions on Nat can be represented in this way (any function provably total in higher order arithmetic [11]), but in practice the codification of a particular function is arbitrarily difficult. Sometimes it is not evident how to describe with Nrec certain functions which can be easily characterized through pattern matching equations. A key example is the boolean function which determines if a natural number is even or not, defined by the following equations:

$$(\mathsf{even}\ 0) = \mathsf{true} \qquad (\mathsf{even}\ (\mathsf{s}\ 0)) = \mathsf{false} \qquad (\mathsf{even}\ (\mathsf{s}\ (\mathsf{s}\ x))) = (\mathsf{even}\ x)$$

The problems of expressiveness derived from the use of elimination constants have been already remarked in [4]. In that paper a different treatment of recursion is suggested in the context of Martin-Löf's type theory. Instead of using elimination constants to codify recursive functions, each function is defined directly by a set of pattern matching equations, as in functional programming languages. In order to overcome the problem of normalization, a recursive definition must pass a syntactical checking before being accepted : the *structurally smaller calls* condition. Nevertheless, this condition does not ensure the termination of recursive functions on certain inductive types, for example those containing a second order quantification [4].

A similar problem between normalization and expressiveness arises when we want to extend the proof system with *coinductive* types, i.e. types whose elements may be infinite objects. As well as an elimination rule can be associated to each inductive type, a *corecursion* rule can be associated to each coinductive one [9]. The aim of corecursion rules is not to eliminate but to generate an element of the type. For example, a rule for generating infinite lists (streams) of booleans can be impredicatively described introducing a primitive constant

$$\mathsf{Scorec} : (X : Set)(X \rightarrow (\mathsf{Bool} \times (\mathsf{Str} + X))) \rightarrow (X \rightarrow \mathsf{Str})$$

Informally, ($\mathsf{Scorec}\ X\ h\ x$) represents the stream generated from $x : X$ by repeatly applying the function $h$. At each step this function provides the boolean in the head of the stream and, either its tail, or another element of $X$ to keep the process. This interpretation justifies the definition of two destructor functions $\mathsf{hd} : \mathsf{Str} \rightarrow \mathsf{Bool}$ and $\mathsf{tl} : \mathsf{Str} \rightarrow \mathsf{Str}$ by the equations

$$\begin{array}{lll}(\mathsf{hd}\ (\mathsf{Scorec}\ X\ h\ x)) = \mathsf{fst}\ (h\ x) & : \mathsf{Bool} & \\ (\mathsf{tl}\ \ (\mathsf{Scorec}\ X\ h\ x)) = s & : \mathsf{Str}, & \text{if } \mathsf{snd}\ (h\ x) = (\mathsf{inl}\ s) \\ \qquad\qquad\qquad\quad = (\mathsf{Scorec}\ X\ h\ y) & : \mathsf{Str}, & \text{if } \mathsf{snd}\ (h\ x) = (\mathsf{inr}\ y)\end{array}$$

The extension of a polymorphic lambda calculus with these rules preserves normalization [7, 9], but describing a stream with Scorec may be even harder than describing a function with Nrec. Consider for example the stream which alternates infinitely the boolean values false and true. While it can be easily described through the equation

$$\mathsf{alter} = (\mathsf{cons}\ \mathsf{false}\ (\mathsf{cons}\ \mathsf{true}\ \mathsf{alter})) \qquad (*)$$

its codification in terms of Scorec leads to a complicated description, which is not evident to do. Furthermore, the basic principles for reasoning with this representation of streams (like the *principle of coinduction* [14, 13]) looks quite complicated.

In [5] the proposal of [4] was complemented introducing a second syntactical criterion (the *guarded induction principle*) for accepting a recursive definition given through equations. The purpose of this condition is to check that definitions like (*) can not be used to construct a term without a head-of-constructor form. As was already said, this is sufficient to ensure the consistency of the system. However, we can not expect for infinite objects like the stream alter to have a (finite) normal form, but just a head-of-constructor one. From this follows that in the approach of [5] the intensional equality of two streams can not be decided, and thus neither the typechecking.

In this paper we formalize an extension of the Calculus of Constructions based on the ideas in [4, 5]. We propose a modification of the structurally smaller calls condition in [4] which does not need to reject recursive types containing second order quantification. We show that the guarded induction principle of [5] does not work for these impredicative types, and we also adapt this second condition. The conversion rules of our system are modified with respect to the ones assumed in the examples of [5] in order to provide a decidable equality for terms. We call this extension $\mathrm{CC}^{(Co)Ind}$.

We also study the relationship between $\mathrm{CC}^{(Co)Ind}$ and a system based on elimination rules for inductive types and corecursion rules for coinductive ones. We develop two mechanical methods for translating a definition verifying the conditions we propose into a term expressed with these kind of rules. The first method can be used to produce a description with Nrec from the equational definition of even. The second method can be applied to the definition of alter in order to obtain its description with Scorec. The existence of these methods of translation provides a justification of the conditions proposed : they ensure that it is possible to explain the definition using safe principles of recursion.

The rest of paper is organized as follows. In section 2 we present the typing rules of $\mathrm{CC}^{(Co)Ind}$ and we comment its main features through some examples. In section 3 we formalize our version of the structurally smaller calls checking, that we call the guarded-by-destructors condition. We explain the method for codifying a guarded-by-destructors function with elimination rules through an example, and we analize the behavior of this codification with respect to conversion. In section 4 we present a counterexample of the guarded induction principle and we propose a modification of it, the guarded-by-constructors condition. We show how to codify a guarded-by-constructors definition using corecursion rules, and we also analyze the behavior of this codification with respect to conversion. In section 5 we draw some conclusions of this work.

## 2 The rules of $\mathrm{CC}^{(Co)Ind}$

The system $\mathrm{CC}^{(Co)Ind}$ is an extension of the Calculus of Constructions (CC) [3]. We use the traditional notation for the terms of CC, writing the abstractions with square brackets and the dependent product of types with curved ones.

$$T ::= Type \mid Set \mid x \mid (x{:}T)T \mid [x{:}T]T$$

The typing rules of these constructions correspond to the GTS [2] with two sorts *Set* and *Type*, one axiom *Set* : *Type*, and where all rules for products are allowed. This basic set of terms will be extended with new constructions for describing recursive types and their introduction rules, a general case expression, and two different fixedpoint operators. In the rest of the section we present the typing rules for these new constructions and the conversion rules associated with them.

### 2.1 Notational Conventions

A typing context is noted with the Greek capital letter $\Gamma$. Metavariables ranging on term variables are noted by lower case letters from the end of the alphabet, like $x$, $w$, $y$, $z$. Terms are represented by capital letters of the alphabet, like $M$, and a list of terms $M_1 \mid M_2 \mid \dots M_n$ by the same symbols in boldface, like $\boldsymbol{M}$. The notation $\boldsymbol{x}$ is used if all the terms of the list are variables. The length of $\boldsymbol{M}$ is written $|\boldsymbol{M}|$, and the term in its $i^{th}$ position $M_i$. The expression $i \in |\boldsymbol{M}|$ means that $i$ is an integer between one and $|\boldsymbol{M}|$. Predicates and functions on terms are noted by calligraphic letters like $\mathcal{P}$ and $\mathcal{F}$, and its application to a term $M$ by curly brackets like in $\mathcal{F}\{M\}$. For the substitution of $N$ to free occurrences of $x$ into $M$ we use the notation $M\{x \leftarrow N\}$. If $\mathcal{P}$ is a property defined on terms, $\mathcal{P}\{\boldsymbol{M}\}$ means that $\mathcal{P}\{M_i\}$ holds $\forall i \in |\boldsymbol{M}|$.

The notation $M = (N\ \boldsymbol{P})$ means that $M$ is the term $N$ if $|\boldsymbol{P}| = 0$ and $(\dots((N\ P_1)\ P_2)\dots P_{|\boldsymbol{P}|})$ otherwise. In the same way, $M = (\boldsymbol{x}{:}\boldsymbol{P})Q$ (resp. $M = [\boldsymbol{x}{:}\boldsymbol{P}]Q$), where $|\boldsymbol{x}| = |\boldsymbol{P}|$, means that $M$ is the term $Q$ if $|\boldsymbol{P}| = 0$ and $(x_1{:}P_1)\dots(x_{|\boldsymbol{x}|}{:}P_{|\boldsymbol{x}|})Q$ (resp. $[x_1{:}P_1]\dots[x_{|\boldsymbol{x}|}{:}P_{|\boldsymbol{x}|}]Q$) otherwise.

Given $M$, it is possible to define an inverse operation which computes the largest list $\boldsymbol{P}$ for which there is a term $N$ such that $M = (N\ \boldsymbol{P})$. Note that there is always such a list. Furthermore, the term $N$ is completely determined by it. The notation $M \equiv (N\ \boldsymbol{P})$ will be used to mean that $N$ and $\boldsymbol{P}$ are the result of applying this destructive operation to $M$. Similarly, $M \equiv [\boldsymbol{x}{:}\boldsymbol{P}]N$ (resp. $M \equiv (\boldsymbol{x}{:}\boldsymbol{P})N$ ) means that $\boldsymbol{x}$ and $\boldsymbol{P}$ are the largest lists for which there is a (unique) term named $N$ such that $M = [\boldsymbol{x}{:}\boldsymbol{P}]N$ (resp. $M = (\boldsymbol{x}{:}\boldsymbol{P})N$).

The expression $P \to Q$ stands for the product $(x{:}P)Q$ where $x$ does not occur free in $Q$.

## 2.2 Recursive Types

The first step in our extension of CC is the introduction of recursively defined types. A recursive type consists of its arity and the signature of its constructors. They are classified into inductive and coinductive ones. When a recursive type is specified, it must be said if it is inductive or coinductive.

*Example 1.* Boolean values, natural numbers, and streams of booleans are all recursive types of arity *Set*, described by the following specifications.

```
Inductive   Bool : Set := true : Bool | false : Bool.
Inductive   Nat  : Set := 0    : Nat  | s     : Nat-> Nat.
Coinductive Str  : Set := cons : Bool -> Str -> Str.
```

For representing recursive types we use the abstract syntax introduced in [12]. The set of terms is extended with two new constructions $T ::= \mathsf{Ind}(x{:}T)\langle \boldsymbol{T}\rangle \mid \mathsf{CoInd}(x{:}T)\langle \boldsymbol{T}\rangle$. In the term $\mathsf{(Co)Ind}(x{:}A)\langle \boldsymbol{C}\rangle$ the subterm $A$ denotes the arity of the type, and the list $\boldsymbol{C}$ its signature of constructors. The variable $x$ is like the carrier of the specification, and it is bound in the terms of $\boldsymbol{C}$. For example, the type Str corresponds to the term $\mathsf{CoInd}(X{:}Set)\langle \mathsf{Bool}{\to}X{\to}X\rangle$. Parametric types are constructed by abstraction of these terms, for instance the type of streams of a given type $A$ corresponds to the term $[A{:}Set]\mathsf{CoInd}(X{:}Set)\langle A{\to}X{\to}X\rangle$. The terms of $\boldsymbol{C}$ must satisfy certain syntactical constraints concerning the occurrences of $x$ inside them. A term satisfying these constraints is called a *form of constructor.*

**Definition 1. (Strictly positive occurrences).** *The variable $X$ occurs strictly positively in the term $P$ if $P \equiv (x : \boldsymbol{M})(X\ \boldsymbol{N})$ and $X$ does not occur free in $M_i$ $\forall i \in |\boldsymbol{M}|$, nor in $N_j$ $\forall j \in |\boldsymbol{N}|$.*

**Definition 2. (Forms of constructor)** *Let $X$ be a variable. The terms which are a form of constructor w.r.t. $X$ are generated by the syntax:*

$$Co ::= (X\ \boldsymbol{N}) \mid P \to Co \mid (x{:}M)Co$$

*with the following restrictions for $X$: it does not occur free in $N_i$ $\forall i \in |\boldsymbol{N}|$, it is strictly positive in $P$, and it does not occur free in $M$.*

**Typing rule for recursive sets.** Consider a term $\mathsf{(Co)Ind}(X{:}A)\langle \boldsymbol{C}\rangle$ such that $\forall i \in |\boldsymbol{C}|$ $C_i$ is a form of constructor w.r.t. $X$. The typing rule for such a term is:

$$\frac{\Gamma \vdash A : Type \qquad \Gamma, X{:}A \overset{\forall i \in |\boldsymbol{C}|}{\vdash} C_i : Set}{\Gamma \vdash \mathsf{(Co)Ind}(X{:}A)\langle \boldsymbol{C}\rangle : A} \qquad [\text{(Co)Ind}]$$

### 2.3 Constructors

A new kind of term $T ::= \mathsf{Constr}(i, T)$ is introduced to represent the constructors of the recursive types. For example, the only constructor cons of Str will be written $\mathsf{Constr}(1, \mathsf{Str})$. In general, the term $\mathsf{Constr}(i, I)$ represents the $i^{th}$ constructor of the recursive type $I$.

**Typing rule for constructors.** Consider a term $\mathsf{Constr}(i, I)$ where $I = \mathsf{(Co)Ind}(X{:}A)\langle \boldsymbol{C}\rangle$ and $i \in |\boldsymbol{C}|$. The typing rule for such a term is:

$$\frac{\Gamma \vdash I : A}{\Gamma \vdash \mathsf{Constr}(i, I) : C_i\{X \leftarrow I\}} \qquad [\text{Constr}]$$

## 2.4 Case Expressions

The second extension is the introduction of a general case expression $T ::=$ <$T$>Case $T$:$T$ of $\langle \boldsymbol{T} \rangle$. This construction works in the same way as the pattern matching expression of the programming language ML, although the syntax is a bit different. The term <$Q$>Case $N$:$P$ of $\langle \boldsymbol{G} \rangle$ defines an element in $Q$ by case analysis on the expression $N$ of (recursive) type $P$. In the most general case $Q$ may be a family of types depending on elements of $P$. The list $\boldsymbol{G}$ corresponds to the cases of the analysis, where each case is an abstraction depending on the variables of the respective pattern of construction.

*Example 2.* The destructors hd and tl on streams are functions defined by case analysis.

```
Definition hd = [s:Str]<Bool>Case  s:Str  of  [b:Bool][t:Str]b.
Definition tl = [s:Str] <Str>Case  s:Str  of  [b:Bool][t:Str]t.
```

Using this notation for case analysis it is easy to describe the rule for typing case expressions on streams:

$$\frac{\Gamma \vdash Q : \mathsf{Str} \to Set \qquad \Gamma \vdash M : \mathsf{Str} \qquad \Gamma \vdash G : (n{:}\mathsf{Nat})(s{:}\mathsf{Str})(Q \ \ (\mathsf{cons}\ n\ s))}{\Gamma \vdash \text{<}Q\text{>}\mathsf{Case}\ \ M{:}\mathsf{Str}\ \ \mathsf{of}\ \langle G \rangle : (Q\ M)}$$

A case expression where $Q$ does not depend on Str can be derived from this rule using a constant predicate, i.e. in the definition of hd the type Bool can be written as the function $[s : \mathsf{Str}]\mathsf{Bool}$. We will use the same notation for dependent and non-dependent case analysis. □

In order to provide a schematic typing rule for any case expression we introduce an operation $\mathcal{S}$ defined by structural induction on the forms of constructor. This operation is used to construct the types of the cases in the expression.

**Definition 3. (Left/right substitution)** *Let $I$, $Q$ and $R$ be three terms and $C$ be a form of constructor w.r.t. $X$. We define a new term $\mathcal{S}\{C, I, Q, R\}$ by induction on the structure of the form of constructor $C$.*

$$\begin{array}{lll} \mathcal{S}\{P \to C, I, Q, R\} & = (y{:}P\{X \leftarrow I\})\mathcal{S}\{C, I, Q, (R\,y)\} & \textit{if } P \equiv (\boldsymbol{x}{:}\boldsymbol{M})(X\ \boldsymbol{N}) \\ \mathcal{S}\{(x{:}M)C, I, Q, R\} & = (x{:}M)\mathcal{S}\{C, I, Q, (R\,x)\} & \textit{if } X \textit{ is not free in } M \\ \mathcal{S}\{(X\ \boldsymbol{N}), I, Q, R\} & = (Q\ \boldsymbol{N}\ R) & \end{array}$$

**Typing rule for case expressions.** Consider a term <$Q$>Case $M$:$S$ of $\langle \boldsymbol{G} \rangle$ where $S \equiv (I\ \boldsymbol{P})$, $I = (\mathsf{Co})\mathsf{Ind}(X{:}A)\langle \boldsymbol{C} \rangle$ with $A \equiv (\boldsymbol{z}{:}\boldsymbol{Z})Set$, and $\boldsymbol{G}$ is a list of terms such that $|\boldsymbol{G}| = |\boldsymbol{C}|$. The typing rule for such a term is :

$$\frac{\Gamma \vdash Q : (\boldsymbol{z}{:}\boldsymbol{Z})(I\ \boldsymbol{z}) \to Set \quad \Gamma \vdash M : S \quad \Gamma \overset{\forall i \in |G|}{\vdash} G_i : \mathcal{S}\{C_i, I, Q, \mathsf{Constr}(i, I)\}}{\Gamma \vdash \text{<}Q\text{>}\mathsf{Case}\ \ M{:}S\ \ \mathsf{of}\ \langle G \rangle : (Q\ \boldsymbol{P}\ M)}$$

**Remark.** It should be noticed that the kind of case analysis proposed here does not correspond to the pattern matching proposed in [4], but rather to the one that can be derived from the usual elimination rule associated to an inductive type. This means that the rule takes no advantage of the information provided by the indexes $\boldsymbol{P}$ of the type $I$ in order to (possibly) eliminate some of the cases.

Thus, in a case analysis of an element with type (Even 0)[2] both the cases of zeven and seven must be provided, even though the index 0 could be used to infer that there is no possibility for the element to be constructed with seven. See [4] for a detailed discussion of pattern matching in the context of dependent types.

## 2.5 Fixedpoint Operators

The last extension is the introduction of two fixedpoint operators for doing recursive definitions, which are named Fix and CoFixThe operator Fix is used to define functions on inductive types, while CoFix is used to define objects in coinductive ones.

*Example 3.* Using these fixedpoints it is possible to provide a direct description of the function even and the stream alter mentioned in the introduction. The three equations defining even are presented in a single term using the case analysis expression.

```
Fix 1  even : Nat -> Bool :=
  [x:Nat]<Nat>Case x:Nat of true |
             [n:Nat]<Nat>Case n:Nat of false | [m:Nat](even m).
CoFix alter : Str := (cons false (cons true alter)).
```

Two different operators are necessary because the reduction rules associated to each one are different (cf. section 2.6). Thus, another two syntactical constructions $T ::= \mathsf{Fix}_k\ f{:}T := T \mid \mathsf{CoFix}\ f{:}T := T$ are introduced. The integer $k$ in $\mathsf{Fix}_k$ points out on which inductive argument of the function the recursion is done. For each operator there is a syntactical condition that should be satisfied as part of its typing rule. These conditions (called $\mathcal{D}$ and $\mathcal{C}$ respectively) constraints the occurrences of $f$ allowed in the body of the operator. We present the typing rules here and discuss the conditions in the next sections.

**Typing rule for Fix.** Let $(\mathsf{Fix}_k\ f{:}B := N)$ be a term such that $B \equiv (\boldsymbol{z}{:}\boldsymbol{Z})(x{:}(I\ \boldsymbol{z}))D$, $|\boldsymbol{z}| = k$ and $I = \mathsf{Ind}(X{:}A)\langle \boldsymbol{C} \rangle$. Let $N \equiv [\boldsymbol{z}{:}\boldsymbol{Z}][x{:}(I\ \boldsymbol{z})]M$. The typing rule of such a term is:

$$\frac{\Gamma \vdash B : Set \qquad \Gamma, f{:}B \vdash N : B \qquad \mathcal{D}\{f, k, x, M\}}{\Gamma \vdash \mathsf{Fix}_k\ \ f{:}B := N\ :\ B} \quad [\text{Fix}]$$

**Typing rule for CoFix.** Let $(\mathsf{CoFix}\ f{:}B := N)$ be a term such that $B \equiv (\boldsymbol{z}{:}\boldsymbol{Z})(I\ \boldsymbol{P})$ and $I = \mathsf{CoInd}(X{:}A)\langle \boldsymbol{C} \rangle$. The typing rule of such a term is:

$$\frac{\Gamma \vdash B : Set \quad \Gamma, f{:}B \vdash N : B \quad \mathcal{C}\{f, N\}}{\Gamma \vdash \mathsf{CoFix}\ f{:}B := N\ :\ B} \quad [\text{CoFix}]$$

## 2.6 Rules of Conversion

We use the symbol $\cong$ for the conversion relation of CC. This relation is extended with rules expressing the congruence w.r.t the new constructions, and also with the following three reduction rules.

[2] See the specification of the type Even in the example of section 3.3.

**Case contraction.** $<Q>\mathsf{Case}\ (\mathsf{Constr}(i,I)\ \boldsymbol{P}){:}S\ \mathsf{of}\ \langle \boldsymbol{G}\rangle \cong (G_i\ \boldsymbol{P})$

**Fix expansion.** Let $F$ be the term $(\mathsf{Fix}_k\ f{:}B := N)$ and $\boldsymbol{P}$ a list of terms of length $k$.

$$(\ F\ \ \boldsymbol{P}\ \ (\mathsf{Constr}(i,I)\ \boldsymbol{Q})\ ) \cong (N\{f \leftarrow F\}\ \ \boldsymbol{P}\ \ (\mathsf{Constr}(i,I)\ \boldsymbol{Q})\ )$$

**CoFix expansion.** Let $F$ be the term $(\mathsf{CoFix}\ f{:}B := N)$.

$$<Q>\mathsf{Case}\ (F\ \boldsymbol{P}){:}S\ \mathsf{of}\ \langle \boldsymbol{G}\rangle \cong <Q>\mathsf{Case}\ (N\{f \leftarrow F\}\ \boldsymbol{P}){:}S\ \mathsf{of}\ \langle \boldsymbol{G}\rangle$$

From the reduction rule of CoFix follows that in example 3 the equalities (hd alter) $\cong$ false and (tl alter) $\cong$ (cons true alter) hold, but the equality alter $\cong$ (cons false (cons true alter)) does not, since the expansion of alter is allowed only when a case analysis of it is done. This is an important difference with respect to the system proposed in [5], where the reduction rules assumed in the examples would correspond rather to $(\mathsf{CoFix}\ f{:}B := N) \cong N\{f \leftarrow (\mathsf{CoFix}\ f{:}B := N)\}$. If alter would be considered as a redex itself, it could not be reduced to a finite normal form, and the property that any term has a normal form is essential for the decidability of typechecking. The price to pay for having a decidable typechecking is that some of the elegant proofs presented in [5] can not be directly translated into $\mathrm{CC}^{(Co)Ind}$, since they use equalities which are not valid in this system[3]. We remark that, even though the terms alter and (cons false (cons true alter)) are not intensionally equal in $\mathrm{CC}^{(Co)Ind}$, they can be proved equal in the sense of Leibniz's propositional equality. A proof of this propositional equality is just the application of the following function unfold of type $(s{:}\mathsf{Str})(s \overset{Str}{=} <\mathsf{Str}>\mathsf{Case}\ s{:}\mathsf{Str}\ \mathsf{of}\ \mathsf{cons})$ to the stream alter. The constant rflex : $(A{:}Set)(a{:}A)a \overset{A}{=} a$ used in its definition is the constructor of the propositional equality type $\overset{A}{=}$.

```
Definition unfold =
  [s:Str]<[x:Str](x=<Str>Case x:Str of cons)>
       Case s:Str of [b:Bool][s:Str](rflex Str (cons b s)).
```

The type of (unfold alter) can be converted into the searched proposition by first performing a CoFix expansion and then a Case contraction :

$$\begin{array}{lcr}
(\mathsf{alter} \overset{Str}{=} <\mathsf{Str}>\mathsf{Case\ alter{:}Str\ of\ cons}) & \cong & \text{[CoFix exp.]} \\
(\mathsf{alter} \overset{Str}{=} <\mathsf{Str}>\mathsf{Case\ (cons\ false\ (cons\ true\ alter)){:}Str\ of\ cons}) & \cong & \text{[Case contr.]} \\
(\mathsf{alter} \overset{Str}{=} \mathsf{(cons\ false\ (cons\ true\ alter))}) & &
\end{array}$$

## 3 The Inductive Operator

We focus now on the fixedpoint operator Fix and the syntactical condition $\mathcal{D}$ it has to satisfy. This condition complements the reduction rule of $\mathsf{Fix}_k$ , ensuring that each expansion of this operator consumes (at least) the constructor in the

[3] See for example the "proof about the list of iterates" described in [5]

head of its (k+1)th argument. Informally, the term $(\mathsf{Fix}_k\ f{:}B := N)$ should satisfy four requirements. First, $f$ may occur in $N$ only at the head of an application. Second, any application of $f$ must be protected by a case analysis of the (k+1)th abstraction of $N$, say $x$. For this reason the condition will be called *to be guarded by destructors*. Third, the (k+1)th argument in the applications of $f$ must be a component of $x$. The (direct) components of $x$ are represented by terms of the form $(z\ \boldsymbol{P})$, with $z$ being a pattern variable of the case analysis which protects the application. Further matchings on these terms allow to access to deeper components of $x$. For example, the definition of even in example 3 verifies these constraints. The only application of even is protected by a case analysis on $x$. The argument of this application is the pattern variable $m$, a deep component of $x$ which is made available by a case analysis of the pattern variable $n$, a direct component of $x$.

Till here, the condition is essentially the same as the *structurally smaller calls* proposed in [4]. However, as was already noticed in that work, these restrictions are not always sufficient to ensure the soundness of the definition. A counterexample can be derived from the introduction of an inductive type V whose specification contains a second order quantification.

```
Inductive Empty  : Set := .
Inductive V      : Set := cnsv : ((A:Set)A->A) -> V.
Fix 1 f:V->Empty :=
  [x:V]<Empty>Case x:V of [h:(A:Set)(A->A)](f (h V (cnsv h)).
```

Using the function f it is possible to construct an element in the empty type, like the non-normalizable term (f (cnsv $[A{:}Set][x{:}A]x$)). We require a fourth condition to avoid this problem: only **recursive** components of $x$ are authorized to be the argument of a recursive call. A pattern variable $y$ representing a component of $x$ is recursive if the type of $x$ occurs in the type of $y$. In the counterexample above $h$ does not represent a recursive component of $x$, since V does not occur in $(A{:}Set)(A \to A)$. Note that what is rejected here is not the type V (which looks perfectly legal in the context of an impredicative type theory like CC) but the definition of f.

## 3.1 Formal Definition of $\mathcal{D}$

The formal description of the guarded-by-destructors condition is provided by a predicate $\mathcal{D}_V\{f,k,x,M\}$ defined by induction on the term $M$. A set of identifiers $V$ is used in the definition of $\mathcal{D}$ to collect the pattern variables in $M$ which represent the recursive components of $x$. A predicate $RP$ allows to distinguish between the recursive and non-recursive components of an object built with $\mathsf{Constr}(i,I)$ from the specification of this constructor. The predicate $\mathcal{D}\{f,k,x,M\}$ in the typing rule [Fix] in section 2 is just $\mathcal{D}_\emptyset\{f,k,x,M\}$.

**Definition 4. (Recursive position)** *Let $C \equiv (\boldsymbol{x}{:}\boldsymbol{M})(X\ \boldsymbol{N})$ be a form of constructor w.r.t. $X$. We say that the number $j$ corresponds to a recursive position of $C$ iff the variable $X$ appears in the term $M_j$. We note this property $RP\{j,C\}$.*

**Definition 5. (Guarded by destructors)** *Let $k$ be a positive integer. Let $M$ be a term, $f$ and $x$ two identifiers and $\mathcal{V}$ a set of identifiers. We present under which conditions $\mathcal{D}_{\mathcal{V}}\{f,k,x,M\}$ holds, depending on the form of $M$. In order to improve the readability we omit the arguments $f$, $k$ and $x$ of $\mathcal{D}$, which are general parameters in this definition.*

- *$f$ does not occur in $M$ : $\mathcal{D}_{\mathcal{V}}\{M\}$ holds without conditions.*
- *$M = [z{:}P]Q$ : $\mathcal{D}_{\mathcal{V}}\{P\}$ and $\mathcal{D}_{\mathcal{V}}\{Q\}$.*
- *$M = (z{:}P)Q$ : $\mathcal{D}_{\mathcal{V}}\{P\}$ and $\mathcal{D}_{\mathcal{V}}\{Q\}$.*
- *$M = (\mathsf{Co})\mathsf{Fix}_{(p)}\ g{:}A := N$ : $\mathcal{D}_{\mathcal{V}}\{A\}$ and $\mathcal{D}_{\mathcal{V}}\{N\}$.*
- *$M = (\mathsf{Co})\mathsf{Ind}(X{:}A)\langle \boldsymbol{C}\rangle$ : $\mathcal{D}_{\mathcal{V}}\{A\}$ and $\mathcal{D}_{\mathcal{V}}\{\boldsymbol{C}\}$.*
- *$M = \texttt{<}Q\texttt{>}\mathsf{Case}\ N{:}S\ \mathsf{of}\langle \boldsymbol{G}\rangle$ : the definition proceeds by case on $N$*
  - *$N \equiv (z\ \boldsymbol{P})$ and $z \in \mathcal{V} \cup \{x\}$ :*
    - *$S \equiv (I\ \boldsymbol{R})$ and $I = \mathsf{Ind}(X{:}A)\langle \boldsymbol{C}\rangle$*
    - *$\mathcal{D}_{\mathcal{V}}\{Q\}$ and $\mathcal{D}_{\mathcal{V}}\{S\}$ and $\mathcal{D}_{\mathcal{V}}\{\boldsymbol{P}\}$*
    - *$\forall i \in |\boldsymbol{G}|$, if we let $C_i \equiv (\boldsymbol{y}{:}\boldsymbol{T})(X\ \boldsymbol{K})$, $G_i \equiv [\boldsymbol{y}{:}\boldsymbol{T}\{X \leftarrow I\}]E$ and $\mathcal{U} = \mathcal{V} \cup \{y_j \mid RP\{j, C_i\}\}$, then $\mathcal{D}_{\mathcal{U}}\{E\}$.*
  - *Otherwise : $\mathcal{D}_{\mathcal{V}}\{Q\}$ and $\mathcal{D}_{\mathcal{V}}\{N\}$ and $\mathcal{D}_{\mathcal{V}}\{S\}$ and $\mathcal{D}_{\mathcal{V}}\{\boldsymbol{G}\}$.*
- *$M \equiv (N\ \boldsymbol{P})$ : the definition proceeds by case on $N$.*
  - *$N = f$ : $|\boldsymbol{P}| > k$, $P_{k+1} \equiv (z\ \boldsymbol{Q})$ with $z \in \mathcal{V}$, and $\mathcal{D}_{\mathcal{V}}\{\boldsymbol{P}\}$.*
  - *Otherwise : $\mathcal{D}_{\mathcal{V}}\{N\}$ and $\mathcal{D}_{\mathcal{V}}\{\boldsymbol{P}\}$.*

## 3.2 Codification of Elimination Rules

This restriction is sufficient to provide a proof system as powerful as one based on elimination rules, like the one in [12]. An elimination rule can be described in $CC^{(Co)Ind}$ as a function containing a Fix definition which is guarded by only one destructor. For instance, the dependent elimination for natural numbers in section 1 can be described by the following function:

```
Definition Nrec =
[Y:Nat->Set][h0:(Y 0)][h1:(m:Nat)(Y m)->(Y (s m))]
  (Fix f :  (n:Nat)(Y n)
         := [n:Nat]<Y>Case n:Nat of h0 | [p:Nat](h1 p (f p))).
```

The property to be proved and the minor premises of the rule are general parameters which are abstracted from the Fix definition. The reduction rules of Fix and Case provide the equalities usually associated to this rule. This codification can be easily generalized to any inductive type, see [8].

## 3.3 Reduction to Elimination Rules

In fact the reciprocal property also holds, i.e. also any Fix function which is guarded by destructors can be described using elimination rules. This means that $CC^{(Co)Ind}$ is a more expressive extension of CC than the one in [12], but (restricted to inductive types) it is essentially equivalent to it. The difficult point to solve in this direction is the codification of those functions doing recursive calls on deep components of its argument, like the function even in example 3. Of course, it is not difficult to describe even with Nrec, for example using the boolean negation function not : Bool $\rightarrow$ Bool:

```
Definition even = [x:Nat](Nrec Bool true [n:Nat][y:Bool](not y) x)
```

But this codification depends too much on the characteristics of this particular function, and can not be extended to other cases. In this section we develop a *general* method of codification, applicable to even as well as to any other function which is guarded by destructors. This method allows us to state the following theorem.

***Theorem 6. (First theorem of equivalence).*** For any function $F$ introduced by a Fix definition there is another function $\widehat{F}$ which is extensionally equal to $F$ and use only elimination rules.

The method we propose can be sketched as follows. Let $F = (\mathsf{Fix}_k\ f{:}B := N)$ and let $B \equiv (z{:}\boldsymbol{Z})(x{:}(I\ z))D$ where $|z| = k$. A new recursive type $I'_D$ : $(z{:}\boldsymbol{Z})(I\ z){\rightarrow}Set$ will be generated from the type $I$ and the family $[z{:}\boldsymbol{Z}][x{:}(I\ z)]D$ (i.e., the codomain of $F$). The codification of $F$ is a function $\widehat{F}$ obtained composing two other functions, that we call *Fbody* and *genI'*.

$$\begin{array}{ll} \mathit{genI'} & : (z{:}\boldsymbol{Z})(x{:}(I\ z))(I'_D\ z\ x) \\ \mathit{Fbody} & : (z{:}\boldsymbol{Z})(x{:}(I\ z))(I'_D\ z\ x) \rightarrow (D\ z\ x) \\ \widehat{F} & = [z{:}\boldsymbol{Z}][x{:}(I\ z)](\mathit{Fbody}\ z\ x\ (\mathit{genI'}\ z\ x)) \end{array}$$

The function *genI'* is defined using both the elimination rule of $I$ and the function *Fbody*. The function *Fbody* is generated performing a syntactical transformation of the body $N$ of $F$. This transformation is defined by induction on the proof that $f$ is guarded by destructors in $N$. We give here the hints for the construction of $I'_D$, *genI'* and *Fbody* through an example. The formal definitions for the general case can be found in [8].

**An Example: the function evod.** The example we consider is a little more complicated version of the function even. Instead of yielding just a boolean saying if the argument is even or not, we would like the function to yield a *proof* that the number is even or odd. Such a proof is an element in one of these inductive families.

```
Inductive Even : Nat->Set :=
 zeven  : (Even 0)  |  seven : (n:Nat)(Even n)->(Even (s (s n))).

Inductive Odd  : Nat->Set :=
 oneodd : (Even (s 0)) | sodd  : (n:Nat)(Odd n)->(Odd (s (s n))).
```

Let us call $EO$ the predicate $[n{:}\mathsf{Nat}](\mathsf{Even}\ n) + (\mathsf{Odd}\ n)$. The function evod : $(n{:}\mathsf{Nat})(EO\ n)$ searched can be introduced by a guarded-by-destructors definition using an auxiliary function lemma : $(n{:}\mathsf{Nat})(EO\ n) \rightarrow (EO(\mathsf{s}\ (\mathsf{s}\ n)))$.

```
Definition lemma  =
[p:Nat][z:(EO p)]<(EO (s (s p)))>Case z:(EO p) of
    [q:(Even p)](inl (seven p q)) | [q:(Odd  p)](inr (sodd  p q)).

Fix 1 evod : (x:Nat)(EO x) :=
[x:Nat]<EO>Case x:Nat of
           (inl zeven) |
           [m:Nat]<[z:Nat](EO (s z))>Case m:Nat of
                                (inr oneodd) | [p:Nat](lemma p (evod p)).
```

Thus, in this example $F$ is the function evod, $I$ is the type of natural numbers, and the family $D$ is $[n$:Nat](Even $n$) + (Odd $n$). Informally, the constructors of $I'_D$ are generated from those of $I$ attaching to each recursive argument another two ones, of type $D$ and $I'_D$ respectively. In the example we have chosen this gives rise to the following type:

```
Inductive Nat' : Nat -> Set :=
    O' : (Nat' O)  |  s' : (x:Nat)(EO x)->(Nat' x)->(Nat' (s x)).
```

An element of Nat' is like a list indexed by its length. At each position this list has a natural number and a proof that this number is even or odd. A particular element of (Nat' $x$) is the list containing the pairs <$y$, (evod $y$)> for all $y < x$. In fact, the purpose of introducing Nat' is to have an auxiliary structure to store the result of some of the "previous calls" of evod. This structure is generated by the following function genNat' : ($n$:Nat)(Nat' $n$).

```
Definition genNat' =
  (Nrec Nat' O' [y:Nat][y2:(Nat' y)](s' y (evodbody y y2) y2) ).
```

No previous calls are necessary to compute (evod 0), thus (genNat' 0) is the empty list $0'$ : (Nat$'$ 0). Assume now that $y^2$ is the list containing the result of evod for all naturals less than $y$. The function evodbody searches into $y^2$ those values necessary to evaluate (evod $y$), and computes this new result. Then $y$ and the value provided by evodbody are added to the list $y^2$ using the constructor s$'$. This yields a list containing the result of evod for all the naturals less than (s $y$). It only remains to construct the function evodbody, which is generated transforming the definition of evod.

```
Definition evodbody =
[x:Nat][x2:(Nat' x)]
 <EO>Case x2:(Nat' x) of
      (inr zeven) |
      [m:Nat][m1:(EO m)][m2:(Nat' m)]<[z:Nat](EO (s z))>
       Case m2:(Nat' m) of
        (inl oneodd) | [p:Nat][p1:(EO p)][p2:(Nat' p)](lemma p p1)
```

Compare the body of evod with the function evodbody. The case analysis of $x$ : Nat in the former was replaced by a case analysis of $x^2$ : (Nat$'$ $x$) in the latter[4]. This change introduced two new pattern variables $m^1$ and $m^2$ in the

[4] This case analysis can be expressed with the elimination rule of genNat'. For simplicity, we keep on using the notation with Case .

case corresponding to s. Then the case analysis of $m$ : Nat in evod was replaced by a case analysis of $m^2$ : (Nat$'$ $m$). Again, this introduced two new pattern variables $p^1$ and $p^2$. Finally, the recursive call on (evod $p$) : ($EO$ $p$) was replaced by the variable $p^1$ : ($EO$ $p$) associated to the argument of the recursive call. The result of this transformation is a non recursive function which computes (evod $x$) from an element of (Nat' $x$). The codification of evod is just the composition of this function with genNat'.

```
Definition evodd^  = [x:Nat](evodbody x (genNat' x) ).
```

## 3.4 Behavior with Respect to Conversion

A natural question to ask is what is the computational behavior provided by this codification. Without giving a formal result, we can say that it is possible to find a set of patterns $\{p_1, \ldots p_n\}$ covering the type $I$ for which $\widehat{F}$ satisfies the same conversion rules than $F$. For the function evod these patterns are $p_1 = 0$, $p_2 = (\mathsf{s}\ 0)$ and $p_3 = (\mathsf{s}\ (\mathsf{s}\ x))$, and the equations

$$(\mathsf{evod}\ 0) \cong (\mathtt{inl}\ \mathsf{zeven}) \qquad (\mathsf{evod}\ (\mathsf{s}\ 0)) \cong (\mathtt{inr}\ \mathsf{oneodd})$$
$$(\mathsf{evod}\ (\mathsf{s}\ (\mathsf{s}\ x))) \cong (\mathsf{lemma}\ x\ (\mathsf{evod}\ x))$$

are also satisfied by $\widehat{\mathsf{evod}}$. We have to state this result with conversions instead of reductions, because the composition (evodbody $x$ (genNat' $x$)) has to be refolded to ($\widehat{\mathsf{evod}}$ $x$) to obtain the third equation. From these conversions follows that evod and its codification behaves in the same way for any closed term of type Nat. Nevertheless, for some open terms, evod satisfies other conversions which its codification does not. For example, the application (evod (s $x$)) is a Fix redex which can be reduced to the term

$$<[z{:}\mathsf{Nat}](EO\ (\mathsf{s}\ z))>\mathsf{Case}\ x{:}\mathsf{Nat}\ \mathsf{of}\ (\mathtt{inr}\ \mathsf{oneodd})\ |\ [p{:}\mathsf{Nat}](\mathsf{lemma}\ p\ (\mathsf{evod}\ p))$$

On the contrary, ($\widehat{\mathsf{evod}}$ (s $x$)) only reduce to a large term containing an (irreducible) case analysis on (genNat' $x$). A pattern like (s $x$) is not enough instantiated to produce the same behavior as evod. Even though this equality does not hold for $\widehat{\mathsf{evod}}$ in the intensional sense, it can be proved in the sense of Leibniz's propositional equality for all $x$ : Nat. The proof proceeds by a simple case analysis of $x$ which unlocks the reduction of (genNat' $x$). Then each case follows straightforwardly by reflexivity.

## 4 The Coinductive Operator

We turn now to the definition of infinite objects using CoFix. In a CoFix definition what is constraint is not the arguments of the recursive calls, but in what context these calls may appear. Recursive calls must be protected by at least one constructor, and no other functions apart from constructors can be applied to them. In particular, nested recursive calls can not occur in the definition. A certain number of case analysis may be used to decide the constructor to be applied, and its arguments. These case analysis may be on terms involving the parameters of the function, but these terms can not contain a recursive call.

*Example 4.* Let us define the stream of booleans (intlv $b$ $s$) obtained interleaving the value $b$ among the elements of the stream $s$ until finding a value true in $s$.

```
CoFix intlv : Bool -> Str -> Str :=
[b:Bool][s:Str]
   <Str>Case s:Str of
         [a:Bool][t:Str]<Str>Case a:Bool of
                       (cons a t) | (cons a (cons b (intlv b t))).
```

This definition verifies the constraints. The only recursive call of intlv is guarded by two constructors, and no other kind of function is applied to it. A case analysis of the stream $s$ and its head $a$ decides when the value $b$ must be inserted. Also the definition of the stream alter in example 3 is guarded by constructors. On the contrary, the following function which filters all the values false from a given stream is not valid.

```
CoFix filter : Str -> Str :=
 [s:Str]<Str>Case s:Str of
                [a:Bool][t:Str]<Str>Case a:Bool of
                          (cons a (filter t)) | (filter t).
```

This definition is not accepted because there is a recursive call of filter which is protected by no constructors. Taking the head of (filter $s$) when $s$ only contains values false would lead to an infinite chain of CoFix expansions. Another example of a wrong definition is the following description of an infinite succession of values false, which allows a recursive call in the argument of a case expression.

```
CoFix ff:Str := (cons false <Str>Case ff:Str of [a:Bool][s:Str]s).
```

In this case is the tail of the stream ff that can not be normalized. □

*Example 5.* We present yet another example to show how CoFix can be used to describe infinite proofs in a clear and simple fashion. It concerns the extensional equality of two streams, a traditional example of a coinductive relation in the literature. This relation can be specified as a coinductive family of types using the propositional equality $\overset{Bool}{=}$ and the destructors hd and tl.

```
Coinductive EqStr : Str -> Str -> Set :=
eqstr : (s1:Str)(s2:Str)
        ((hd s1)=(hd s2))->(EqStr (tl s1) (tl s2))->(EqStr s1 s2).
```

Consider now these two new versions of a process generating a sequence which indefinitely alternates the booleans false and true:

```
CoFix gen1:Bool->Str := [b:Bool](cons  b       (gen1 (not b))).
CoFix gen2:Bool->Str := [b:Bool](cons (not b) (gen2 (not b))).
```

Both definitions are guarded by constructors. Let us call $\text{alter}_1$ the stream ($\text{gen}_1$ false) and $\text{alter}_2$ the stream ($\text{gen}_2$ true). A proof of the extensional equality between $\text{alter}_1$ and $\text{alter}_2$ is an infinite object of type (EqStr $\text{alter}_1$ $\text{alter}_2$), which can be easily defined using CoFix as follows.

```
CoFix eqprf : (EqStr alter1 alter2) :=
   (eqstr alter1 alter2 (rflx Bool false)
      (eqstr (tl alter1) (tl alter2) (rflx Bool true) eqprf)).
```

The definition is valid since the recursive call of eqprf is protected by two applications of the constructor eqstr. □

This syntactical criterion for accepting the definition of an infinite object was first proposed in [5]. However, there is a problem —dual of the one mentioned for the structurally smaller calls— which makes this condition unsufficient to ensure consistency in the context of impredicative types. Consider the following counterexample:

```
Coinductive U : Set := cnsu : (A:Set)(A->Empty)->A->U.
Definition  g : U -> Empty =
   [x:U]<Empty>Case x:U  of  [A:Set][h:A->Empty][a:A](h a).
CoFix  u : U := (cnsu U g u).
```

Even though u is protected by the constructor consu, it can be used to find an element in the empty type, like the non-normalizable term (g u). As in the inductive case, we propose a stronger constraint to avoid these undesired definitions: a recursive call may occur only in the **recursive** components of the object which is being defined. Note that this is not the case in the example above, where u appears as a component of itself, but not as a recursive one.

In $CC^{(Co)Ind}$ a recursive definition is also a term, so there is in principle the possibility of nesting definitions. Nevertheless, we remark that this possibility poses new problems for the condition. A Fix definition can not appear inside a CoFix one, as follows from the following counterexample:

```
CoFix g : Nat->Str :=
 (Fix 1 f : Nat->Str :=
    [x:Nat](cons x <Str>Case x:Nat of
                         (g (s x)) | [m:nat](tl (tl (f m)))) ).
```

Note that although the function f does only structurally smaller calls, and the only application of g is guarded by one constructor, the term (tl (g (s 0)) has no normal form. It is less clear that nested occurrences of two CoFix definitions should be forbidden. Neither these latter kind of definitions will be allowed in our condition, but we have to point out that we do not know of any counterexample for them.

## 4.1 Formal Definition of $\mathcal{C}$

The guarded-by-constructors condition is formally described using two predicates $\mathcal{C}_0\{f, M\}$ and $\mathcal{C}_1\{f, M\}$, both defined by induction on the term $M$. The predicate $\mathcal{C}_0$ ensures that, going downwards in $M$, after a certain number of abstractions and case analysis, there is always a constructor. Once a constructor has been placed, the predicate $\mathcal{C}_1$ allows to use $f$ in its recursive arguments. The condition $\mathcal{C}\{f, M\}$ in the typing rule [CoFix] of section 2 is just $\mathcal{C}_0\{f, M\}$.

**Definition 7. (Guarded by constructors)** *Let $M$ be a term and $f$ be an identifier. We present under which conditions the predicates $\mathcal{C}_h\{f, M\}$ $(h = 0, 1)$ hold, depending on the form of $M$.*

- *$M = [x{:}P]N$ : $f$ does not occur free in $P$ and $\mathcal{C}_h\{f, N\}$.*
- *$M = $ <Q>Case $N{:}P$ of $\langle \boldsymbol{G}\rangle$ : $Q \equiv [\boldsymbol{x}{:}\boldsymbol{T}](I\ \boldsymbol{K})$, where $I$ is a coinductive type, there are no free occurrences of $f$ in $N$, $Q$ nor $P$ and $\mathcal{C}_h\{f, \boldsymbol{G}\}$.*
- *$M \equiv (\mathsf{Constr}(i, I)\ \boldsymbol{P})$ : $I = \mathsf{CoInd}(X{:}A)\langle \boldsymbol{C}\rangle$ and $\forall j \in |\boldsymbol{P}|$ if $RP\{j, C_i\}$ then $\mathcal{C}_1\{f, P_j\}$ and otherwise $f$ does not occur free in $P_j$.*
- *$M \equiv (f\ \boldsymbol{P})$ : $h = 1$ and $\forall j \in |\boldsymbol{P}|$ $f$ does not occur free in $P_j$.*
- *Otherwise : $h = 1$ and $f$ does not occur free in $M$.*

It could be also added (as in [5]) that $\mathcal{C}_0\{f, M\}$ holds when $f$ does not occur free in $M$. This would not introduce really new definitions. If $f$ does not occur free in $M$, then $M$ can be replaced by a case analysis of itself whose cases consists in the application of the respective constructor. This transformation yields a term equivalent to $M$ which verifies the previous definition of $\mathcal{C}_0$.

## 4.2 Corecursion Rules

The way of describing infinite objects explained above is well known from lazy functional programming languages, but it is rather new in the context of proof systems based on typed lambda calculus. Most of the approaches for extending typed lambda calculus with coinductive types are based on the interpretation of these types as the greatest fix point (gpf) of certain monotonic operator $T$ [10, 7]. The type of streams, for example, can be interpreted as the gfp of $T(X) = \mathsf{Bool} \times X$. A gfp of $T$ can be codified in an impredicative typed lambda calculus (for example Girard's system F) as the type $I = \exists X.(X{\rightarrow}T(X)) \times X$ [9]. This corresponds to Tarski's description of a gfp as the union of all $X$ verifying $X \subseteq T(X)$. From this description it is possible to find a proof that $I$ is in fact a gfp of $T$, that is, a function $Icoiter : \forall X.(X{\rightarrow}T(X)){\rightarrow}(X{\rightarrow}I)$. The term $(Icoiter\ X\ h\ x)$ can be interpreted in an operational sense as an iterative process which generates an object of $I$ by repeatly applying the function $h$ from the value $x$. Under this vision, the constructors of $I$ are not primitive expressions, but they are defined in terms of $Icoiter$. This codification of constructors does not provide the expected computational behavior with respect to case analysis. This is shown for example in [7], where a stronger (corecursive) rule $Icorec$ : $\forall X.(X{\rightarrow}T(I + X)){\rightarrow}(X{\rightarrow}I)$ is proposed to overcome the problem. The rather limited experience in the development of mechanized proofs concerning infinite objects is based on the use of such *corecursion rules* [13, 9]. Therefore, it is important to clarify the relationship between CoFix definitions and these rules.

## 4.3 Representation of Corecursion Rules

In $\mathrm{CC}^{(Co)Ind}$ it is possible to codify a corecursion rule in the style of [7] using a CoFix definition which is guarded by only one constructor. For example, the corecursion rule for streams described in section 1 (and used in the experiment of [9]) can be represented by the following function Scorec.

```
Definition Scorec =
[X:Set][h:X -> Bool*(Str+X)]
 (CoFix f : X -> St :=
   [x:X]<Str>Case (h x) : Bool*(Str+X) of
     [a:Bool][z:Str+X](cons a <Str>Case z:Str+X of [s:Str]s | f)).
```

In order to describe the corecursion rule of $I$ the operator $T(X)$ defining $I$ must be made explicit. This is one of the facts which makes these rules quite tedious. This operator can be represented as an inductive type which depends on a general parameter $X$: if $I = \mathsf{CoInd}(X{:}A)\langle \boldsymbol{C}\rangle$, then the operator $T_I$ is $[X{:}A]\mathsf{Ind}(Y{:}A)\langle \boldsymbol{C}^*\rangle$, where $C_i^* = (\boldsymbol{x}{:}\boldsymbol{M})(Y\ \boldsymbol{N})$ if $C_i = (\boldsymbol{x}{:}\boldsymbol{M})(X\ \boldsymbol{N})$. Thus, $(T_{St}\ X){=}\mathsf{Ind}(Y{:}Set)\langle\mathsf{Bool}{\rightarrow}X{\rightarrow}Y\rangle$, which is just the specification of $\mathsf{Bool} \times X$. If we note $B_1{\oplus}B_2$ the term $[z : \boldsymbol{Z}]((B_1\ \boldsymbol{z}) + (B_2\ \boldsymbol{z}))$ for $B_1$ and $B_2$ two types of the same arity $A \equiv (\boldsymbol{z} : \boldsymbol{Z})Set$, the corecursion rule associated to $I$ is a function $Icorec : (Y{:}A)((\boldsymbol{z}{:}\boldsymbol{Z})(Y\ \boldsymbol{z}) \rightarrow (T_I\ I{\oplus}Y\ \boldsymbol{z})) \rightarrow (\boldsymbol{z}{:}\boldsymbol{Z})(Y\ \boldsymbol{z}) \rightarrow (I\ \boldsymbol{z})$. The method for generating such a function from the specification of $I$ can be found in [8]. Note that taking $I$ as the predicate EqStr of example 1, the associated corecursion rule is a function

$$\begin{aligned}\mathsf{Eqcorec} : (Y{:}\mathsf{Str}{\rightarrow}\mathsf{Str}{\rightarrow}Set)\\ &((z_1{:}\mathsf{Str})(z_2{:}\mathsf{Str})(Y z_1\ z_2){\rightarrow}(\mathsf{hd}\ z_1){=}(\mathsf{hd}\ z_2) \wedge (\mathsf{EqStr}{\oplus}Y\ (\mathsf{tl}\ z_1)\ (\mathsf{tl}\ z_2)))\\ &\quad{\rightarrow}(z_1{:}\mathsf{Str})(z_2{:}\mathsf{Str})(Y z_1\ z_2){\rightarrow}(\mathsf{EqStr}\ z_1\ z_2)\end{aligned}$$

which expresses the so-called *principle of (strong) coinduction* [14, 13]. This proof principle states that two particular streams $s_1$ and $s_2$ are equal if there exist at least one *bisimulation* satisfied by these streams. A bisimulation is a binary relation $Y$ on streams such that any pair of streams satisfying $Y$ must have equal heads, and tails also satisfying $Y$. The strong version of this principle used in the experiments in [13] also admits the tails to be directly equal if they do not satisfy the bisimulation. We invite the reader to reformulate the proof in example 5 using Scorec and EqStcorec and verify how tedious is the use of corecursion rules, even in simple cases like that.

## 4.4 Reduction to Corecursion Rules

The use of CoFix definitions provides an easier way for describing infinite objects than the use of corecursion rules, but not more strength. As we shall see in this section, any infinite object defined with CoFix can be also described using corecursion rules.

Describing a stream like alter with Scorec poses the dual problem of describing a function like even with Nrec. The rule Nrec is thought to describe functions which consumes one constructor at a time, but even consumes two. Symmetrically, the rule Scorec generates a stream providing one boolean at a time, while alter provides two in a single step. Again, the challenge is to find a general method of codification, which can be used to describe with Scorec both the stream alter as well as, for example, the function intlv of example 4. We propose a method which can be used to describe any CoFix definition which is guarded by constructors. It allows us to state this second result.

***Theorem 8. (Second theorem of equivalence).*** For any infinite object generated by a CoFix function $F$ there is another function $\widehat{F}$ described only with corecursive rules which generates an object extensionally equal to the first.

Let $F = (\mathsf{CoFix}\ f{:}B := N)$ and $B \equiv (w{:}\boldsymbol{Q})(I\ \boldsymbol{U})$. We are going to codify $F$ in terms of *Icorec*, the corecursion rule associated to $I$. We follow a symmetrical strategy to the one used for Fix functions, introducing a new recursive type $I'$ with the same arity as $I$. This type is generated from $I$ and from the family $[z{:}\boldsymbol{Z}]\sum_{w:\boldsymbol{Q}}.(z = \boldsymbol{U})$, where $\sum$ represents the dependent sum of types. The codification $\widehat{F}$ of $F$ is the composition of two functions that we call —similarly to the inductive case— *genI* and *Fbody*.

$$\begin{aligned} genI &: (z{:}\boldsymbol{Z})(I'\ z) \rightarrow (I\ z) \\ Fbody &: (w{:}\boldsymbol{Q})(I'\ \boldsymbol{U}) \\ \widehat{F} &= [w{:}\boldsymbol{Q}](genI\ \boldsymbol{U}\ (Fbody\ w)) \end{aligned}$$

The function *genI* is expressed in terms of *Icorec* and *Fbody*. The function *Fbody* is obtained through a syntactical transformation of the term $N$, defined by induction on the proof that $f$ is guarded by constructors in $N$. Again, we describe here the construction of $I'$, *genI* and *Fbody* using an illustrative example, and refer the reader to [8] for the general case.

**An Example : the function intlv.** Let us take $F$ to be the function intlv : Bool $\rightarrow$ Str $\rightarrow$ Str in example 4. Then the coinductive type $I$ is Str. Since Str has arity $Set$, the family used to generate the type Str$'$ is just the product Bool $\times$ Str. We rename this type as Cls, because in a certain sense its constructor will be used to *enclose* the arguments of intlv. Informally, the constructors of $I'$ are generated from those of $I$ introducing the possibility of choosing among an element of $I$, *Cls* or $I'_B$ at each recursive argument. In this particular case, this gives rise to the following types:

```
Inductive Cls  : Set =  cls    : Bool -> Str -> Cls.
Inductive Str' : Set =  cons'  : Bool -> (Str+Cls+Str') -> Str'.
```

**Remark.** The specification of cons$'$ does not correspond to a form of constructor as they were defined in section 2. Obviously, this type is isomorphic to an other type which has three constructors, one for each possibility in the second argument of cons$'$. Nevertheless, in the general case, the description of $I'$ needs a wider notion of strict positivity than the one proposed in definition 1. In particular, we need the usual consideration that $X$ is positive in $A + B$ if it is positive both in $A$ and $B$[5]. Anyway, as we shall see, the translation method does not need to do recursion on $I'$. Thus, the definition of this type can not lead us to introduce non normalizable terms, which is the purpose of constraining the forms of constructor.

An element of Str$'$ is like a finite list of booleans, ended either by a stream or by a "closure", containing the arguments needed to generate a stream with

[5] We prefer not to include this case in definition 1 since it would complicate a lot the conditions $\mathcal{D}$ and $\mathcal{C}$, as well as the reduction to elimination and corecursion rules.

intlv. The purpose of introducing this type is to represent finite segments of the stream (intlv $b$ $s$). An element of Str$'$ representing such a segment can be obtained performing a syntactical transformation of the definition of intlv. This transformation consists in four replacements. Let us call $in_1$,$in_2$ and $in_3$ the three constructors of the sum Str + Cls + Str$'$. First, the outermost applications of cons are replaced by applications of cons$'$. Second, the inner applications of cons are replaced by applications of cons$'$, and then injected into Str$'$ with $in_3$. Third, the applications of intlv are replaced by applications of cls, and then injected into Cls with $in_2$. Finally, the recursive arguments of cons applications in which intlv does not occur free are injected into Str with $in_1$. This yields the following function intlvbody : Bool $\rightarrow$ Str $\rightarrow$ Str$'$:

```
Definition intlvbody =
[b:Bool][s:Str]
<Str'>Case s:Str of
  [a:Bool][t:Str]<Str'>Case a:Bool of
    (cons' a (in1 t)) | (cons' a (in3 (cons' b (in2 (cls b t))))).
```

The stream itself is generated with Scorec from the segments provided by intlvbody. Each segment is copied to the output, one boolean at a time, until a whole stream or a closure is found. If the segment finishes with a stream, the process stops yielding this stream as the whole rest. If it finishes with a closure, a new segment is generated applying intlvbody to the values in the closure, and the procedure continues in the same way with this new segment. This process is performed by the function genStr : Str$'$ $\rightarrow$ Str.

```
Definition H =
[x:Str']<Bool*(Str+Str')>
  Case x : Str' of
     [a:Bool][y:(Str+Cls+Str')]
       (pair a  <Str+Str'>Case y : (Str+Cls+Str') of
                     [y1:Str](in1 y1)                            |
                     [y2:Cls](in2 <Str'>Case y2:Cls of intlvbody) |
                     [y3:Str'](in2 y3)      ).

Definition genSt = [x:Str'](Scorec Str' H x).
```

The initial segment necessary to start the process genStr is also provided by intlvbody. Thus, the representation of intlv is just the composition of genStr and intlvbody.

```
Definition intlv^ = [w1:Bool][w2:Str](genStr (intlvbody w1 w2)).
```

Using the same method the definition of eqprf in example 5 can be transformed into a proof based on the coinduction principle. The family EqStr$'$ corresponds to the bisimulation of the proof, and the constant eqprfbody to the proof that alter$_1$ and alter$_2$ are in the bisimulation. Usually, the difficult part in the proofs by coinduction is to find the suitable bisimulation [9, Conclusions]. It is a surprising fact that this relation can be generated automatically from a much more simple description of the proof which does not mention it explicitly.

### 4.5 Behavior with Respect to Conversion

This codification provides a result symmetrical to the one pointed out in section 3.4. In this case it is possible to find a set of *destructor* operations $\{d_1, \ldots d_n\}$ satisfying the same conversions for an element generated either by $F$ or by $\widehat{F}$. Consider for example the stream generated applying intlv to false and alter, and the destructor operations $d_1 = \mathsf{hd}$, $d_2 = (\mathsf{hd} \circ \mathsf{tl})$ and $d_3 = (\mathsf{tl} \circ \mathsf{tl})$. These destructors are sufficient to select any boolean of the stream. They satisfy the following conversions:

$$(d_1\ (\mathsf{intlv\ false\ alter})) \cong \mathsf{false} \qquad (d_2\ (\mathsf{intlv\ false\ alter})) \cong \mathsf{false}$$
$$(d_3\ (\mathsf{intlv\ false\ alter})) \cong (\mathsf{intlv\ false\ (cons\ true\ alter)})$$

which are also valid for $\widehat{\mathsf{intlv}}$. Nevertheless, as in the inductive case, the original function verifies more conversions than its codification. Consider for example the destructor tl. The application (tl (intlv false alter)) is a CoFix redex, which can be reduced to the normal form (cons false (intlv false (cons true alter))). However, an irreducible application of genStr blocks the reduction of (tl ($\widehat{\mathsf{intlv}}$ false alter)) to this value. This is the dual situation of the application of evod to the pattern (s $x$), mentioned in section 3.4: the operation tl does not destruct enough the stream to obtain the same computational behavior. As in that case, the equation which does not hold in the intensional sense can be proved in the propositional sense using Leibniz's equality. The proof follows straightforwardly from the application of the function unfold in section 2 to the term (tl ($\widehat{\mathsf{intlv}}$ false alter)).

## 5 Conclusions and Further Work

We have formalized an extension of the Calculus of Constructions which allows a more direct description of recursive definitions. This way of describing recursive definitions is as powerful as the one based on elimination and corecursion rules. We have also show how it can be explained in terms of these rules.

Known mechanized experiments with coinductive types are based on a complicated encoding of greatest fix points and corecursive operators [13, 9]. The approach introduced in [5] offers an alternative way, which leads to much simpler proofs. We think that the codification of CoFix definitions proposed provides a better understanding about the relationship between this new approach and the traditional consideration of coinductive types as greatest fix points. In fact, it was this codification which led us to detect some problems in the guarded induction principle of [5] and showed how to overcome them. Peter Aczel pointed out to us its similarity with the Solution Lemma of his axiomatic theory of non-well founded sets [1]. It would be interesting to investigate further if this codification could be useful for relating the approach in [5] with Aczel's work. The dual interpretation of inductive types as least fix points suggested that it should exist a dual codification for Fix definitions with elimination rules.

The next step in our work will be to experiment with this extension in the proof environment Coq [6]. A fixedpoint operator in the style of Fix has been already implemented by Christine Paulin for the version V5.10 of the system, and there is an experimental version with coinductive types that will be made available soon. The reduction rule proposed for CoFix overcomes one of the major problems to mechanize the approach of [5], i.e. the existence of terms without a finite normal form. Even though this rule makes some of the smart proofs in [5] no longer valid, it seems that they can be easily reformulated using a propositional version of the equalities loosed. A future point to study is the possibility of generating a proof of these equalities (like the function unfold in section 2) from the CoFix definition. It also rests some work to do in order to understand what is the better way to integrate the guard checking to the interactive proof editor of Coq.

**Acknowledgment.** I am greatly indebted to Christine Paulin-Mohring for many comments about previous versions of this paper. Thanks also to Thierry Coquand for his helpful remarks about his work.

## References

1. Peter Aczel. *Non-Well-Founded Sets*, volume 14 of *CLSI Lecture Notes*. 1988.
2. H. Barendregt. Lambda Calculi with types. Technical Report 91-19, Catholic University of Nijmegen, 1991.
3. T. Coquand. Metamathematical Investigations of a Calculus of Constructions. In P. Odifreddi, editor, *Logic and Computer Science*, volume 31 of *The APIC series*, pages 91–122. Academic Press, 1990.
4. Thierry Coquand. Pattern-Matching in type theory. In Bengt Nordström, Kent Petersson, Gordon Plotkin, editor, *Informal Proceedings of the 1992 Workshop on Types for Proofs and Programs*, pages 71–84, 1992.
5. Thierry Coquand. Infinite objects in Type Theory. In Henk Barendregt, Tobias Nipkow, editor, *Types for Proofs and Programs*, pages 62–78. LNCS 806, 1993.
6. G. Dowek et al. The Coq Proof assistant user's guide - Version 5.8. Technical Report 154, INRIA Roquencourt, 1993.
7. Herman Geuvers. Inductive and Coinductive types with iteration and recursion. In Bengt Nordström, Kent Petersson, Gordon Plotkin, editor, *Informal Proceedings of the 1992 Workshop on Types for Proofs and Programs*, pages 193–217, 1992.
8. E. Giménez. Codifying guarded definitions with recursive schemes. Technical report, Laboratoire de l'Informatique du Parallélisme, ENS-Lyon, 1994.
9. Francois Leclerc and Christine Paulin-Mohring. Programming with streams in Coq. A case study : the Sieve of Eratosthenes. In Henk Barendregt, Tobias Nipkow, editor, *Types for Proofs and Programs*, pages 191–212. LNCS 806, 1993.
10. F.P. Mendler. *Inductive Definitions in Type Theory*. PhD thesis, Cornell University, 1987.
11. Christine Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. PhD thesis, Université Paris VII, 1989.
12. Christine Paulin-Mohring. Inductive definitions in the system Coq : Rules and Properties. In M. Bezem, J.F. Groote, editor, *Proceedings of the TLCA*, 1993.
13. Lawrence Paulson. Co-induction and Co-recursion in Higher-order Logic. Technical report, Computer Laboratory, University of Cambridge, 1993.
14. Andrew M. Pitts. A co-induction principle for recursively defined domains. Technical Report 252, Computer Laboratory, University of Cambridge, April 1992.