RAIRO-Theor. Inf. Appl. 43 (2009) 703–766
DOI: 10.1051/ita/2009015

Available online at:
www.rairo-ita.org

# TWO EXTENSIONS OF SYSTEM F WITH (CO)ITERATION AND PRIMITIVE (CO)RECURSION PRINCIPLES*

Favio Ezequiel Miranda-Perea[1]

**Abstract.** This paper presents two extensions of the second order polymorphic lambda calculus, system F, with monotone (co)inductive types supporting (co)iteration, primitive (co)recursion and inversion principles as primitives. One extension is inspired by the usual categorical approach to programming by means of initial algebras and final coalgebras; whereas the other models dialgebras, and can be seen as an extension of Hagino's categorical lambda calculus within the framework of parametric polymorphism. The systems are presented in Curry-style, and are proven to be terminating and type-preserving. Moreover their expressiveness is shown by means of several programming examples, going from usual data types to lazy codata types such as streams or infinite trees.

**Mathematics Subject Classification.** 03B40, 68N18, 68Q42, 68Q65.

## 1. Introduction

The second-order polymorphic lambda calculus, system F, is a very expressive type system, invented independently by Girard in 1972 and Reynolds in 1974, coming from entirely different motivations. Girard (see [9,10]) was pursuing an extension of the well-known Curry-Howard correspondence with universal quantifiers ranging over propositions; whereas, Reynolds (see [31]) was seeking an extension

*Keywords and phrases.* Coiteration, corecursion, iteration, primitive recursion, system F, monotone inductive type, monotone coinductive type, monotonicity witness, saturated sets, algebras, coalgebras, dialgebras.

* *This research was partially supported by México's National Council of Science and Technology Conacyt, with postdoctoral grant number 50289.*

[1] Departamento de Matemáticas, Facultad de Ciencias UNAM, Circuito Exterior S/N. Ciudad Universitaria, 04510 México, D.F. México; favio@ciencias.unam.mx

Article published by EDP Sciences

of common typed programming languages to permit passing types as parameters, that is, to allow the definition of so-called polymorphic procedures that could accept arguments of a variety of types. This is known nowadays as parametric polymorphism and has become a common feature of mainstream programming languages due to several reasons, for instance, it allows the possibility of writing generic programs, *i.e.* programs applicable in a number of different contexts, by associating all behavior of parameter values with the types of parameters. This increases the flexibility, re-usability and expressive power of a programming environment; an important feature for software engineering where a key goal is to support the production and use of reusable code. To guarantee that the code is reused in a correct way, strong typing is desirable. Genericity in code can best be expressed by polymorphic types which, can either be stated explicitly (*à la Church*), or inferred (*à la Curry*). System F is also popular as a highly-expressive typed intermediate language employed in the design of compilers for functional languages like Haskell, in particular the GHC compiler appears to be the first to use system F as a basis. Its use avoids the need for downcasting and allows a compiler to find more programming errors due to its ability to verify the type correctness in compile time (*i.e.* syntactically). These are only some reasons to use a type system similar to (a fragment of) F in real world programming languages, for example it is embodied in the type system of ML. From the theoretical point of view system F has also great benefits: parametricity has proven to be a powerful principle for establishing abstraction properties, proving equivalence of programs and inferring useful properties of programs from their types alone (see [39]); an important achievement in the theory of programming languages is the ability to capture the generic nature of a program by means of a term having a polymorphic type. On the other hand, system F is a strongly terminating language, which means that every program describes a terminating computation, an important theoretical property which implies that the language lacks general recursion and therefore only total functions can be expressed. This could lead us to think that the language is not suitable for practical computation; however, almost everything that one might want to compute can be expressed in F. In particular all usual data types in computer science, like tuples, lists, or trees, have a description within the system as inductive types [2]. Moreover, lazy data types like streams or infinite trees can also be defined as coinductive types [40]. However this representations lack efficiency, since it is well-known that the native recursion operators of (co)inductive types encompass only (co)iteration, that is, the impredicative encoding of (co)inductive types only models (co)iteration and not primitive (co)recursion. In particular, the implementation of the predecessor function on natural numbers traverses a whole numeral in order to remove one constructor; and therefore does not run on constant time as in the case of a primitive recursive implementation. This has lead to investigate extensions of system F with (co)iteration principles to avoid the impredicative encodings, but also with primitive (co)recursion principles. These extensions are not definitional, since it is believed that primitive recursion cannot be computationally reduced to iteration in a faithful way. Such conjecture originates from the fact that no definition of

natural numbers in system F supporting a constant-time implementation of the predecessor seems possible, as discussed in [33].

In this paper we develop two such extensions by means of the categorical interpretation of functional programming, where inductive (coinductive) types are defined as initial algebras (final coalgebras) of functors. This categorical view allows to extend a type system with syntactical functors and primitive constructors for (co)iteration and (co)recursion. This way, the introduction rule of an inductive type works as a data constructor and its elimination rules as recursion principles; the introduction rules for a coinductive type work as corecursion schemes, and its elimination rule as codata destructor. This kind of systems has been known since the work of Hagino [13] for (co)iteration and Geuvers [7] for primitive (co)recursion; since then, several related systems have been described, we can find extensions of the simply typed lambda calculus with positive (co)inductive types and (co)iteration [12,13], extensions of system F with either, positive or monotone inductive types including iteration and/or primitive recursion [19,20,25,26], or with coinductive types, including coiteration and/or primitive corecursion [20]. More recently, in [1] we can find several systems of (co)inductive constructors of higher kinds (higher-order nested datatypes) with (co)iteration principles, which all happen to be definable within the system $\mathsf{F}^{\omega}$ of higher-order parametric polymorphism. However, to the best of our knowledge, the taxonomy of (co)inductive type systems lacks a system including monotone (co)inductive types, polymorphism, primitive (co)recursion and inversion principles. In this paper we present two such systems using the typing *à la Curry*. The first handles conventional (co)inductive types modeling (co)algebras, whereas the second models dialgebras, in the same way as in [13], by means of clausular[1] (co)inductive types. Both systems are terminating and preserve types, a non-trivial property in Curry-style systems. Moreover, they are full monotone, that is, there is no positivity restriction in the construction of a (co)inductive type.

### 1.1. Overview of the paper

After mentioning some preliminaries on category theory and the polymorphic lambda calculus, we develop our first system called MICT, we give some examples of programming and directly prove the termination (strong normalization) of the operational semantics by means of an extension of the usual method of saturated sets for (co)inductive types. In particular the constructions on saturated sets for coinductive constructions that we present are new. In Section 5 we develop a second system, MCICT, which enhances the former by allowing definitions with several constructors/destructors, feature which is illustrated in several examples. Safety for this system is proven by embedding it into the previous system to ensure termination and by proving directly its type-preservation. Finally, we point to some future and related work.

[1] The name "clausular" is mine.

## 2. Preliminaries

In this section we recall some categorical concepts as well as our base type system, the second order polymorphic lambda calculus $\mathsf{F}$.

### 2.1. System $\mathsf{F}$

Our basic framework is the well-known system $\mathsf{F}$ of Girard [11] and Reynolds [31] in Curry-style flavor. For ease of presentation we include sum and product types as primitive constructors.

- Types built from an infinite set of type variables denoted by $X$.

$$A, B, C, F, G ::= X \mid A \to B \mid \forall X.A \mid A + B \mid A \times B.$$

- Terms built from an infinite set of term variables denoted by $x$.

$$t, r, s \quad ::= \quad x \mid \lambda x.r \mid rs \mid \mathsf{inl}\, r \mid \mathsf{inr}\, s \mid \mathsf{case}(r, x.s, y.t) \mid \langle r, s\rangle \mid \mathsf{fst}\, r \mid \mathsf{snd}\, r.$$

The dot notation on universal types, lambda abstractions, case analysis and any other constructor appearing later, denotes binding: in an expression $\forall X.A$ the occurrences of the type variable $X$ in $A$ are bound. Analogously the occurrences of the term variable $x$ in $r$ are bound in $\lambda x.r$, and similarly those of $x$ in $s$ and of $y$ in $t$ in $\mathsf{case}(r, x.s, y.t)$. This binding mechanism using the dot avoids the use of parentheses, the dot signals an opening parentheses which closes as far to the right as syntactically possible. We will drop the dot if the expression after it consists of a single symbol, writing $\lambda xx$ instead of $\lambda x.x$ for instance. We take also the liberty of omitting parentheses as much as possible, in particular we assume that applications associate to the left, writing $rs_1 \dots s_n$ for $(\dots(rs_1)s_2)\dots s_n)$.

- Contexts are sets of the form $\Gamma = \{x_1 : A_1, \dots, x_n : A_n\}$. The expression $\Gamma, x : A$ denotes the context $\Gamma \cup \{x : A\}$ always assuming that $x$ was not previously declared in $\Gamma$. The set $FV(\Gamma)$ of free type variables of $\Gamma$ is defined as usual.
- Typing rules of the form $\Gamma \vdash t : A$ denoting that $t$ is a well-formed term of type $A$ in context $\Gamma$.

$$\frac{}{\Gamma, x : A \vdash x : A}\ (\text{Var})$$

$$\frac{\Gamma, x : A \vdash r : B}{\Gamma \vdash \lambda x.r : A \to B}\ (\to I) \qquad \frac{\Gamma \vdash r : A \to B \quad \Gamma \vdash s : A}{\Gamma \vdash rs : B}\ (\to E)$$

$$\frac{\Gamma \vdash t : A \quad X \notin FV(\Gamma)}{\Gamma \vdash t : \forall X.A}\ (\forall I) \qquad \frac{\Gamma \vdash t : \forall X A}{\Gamma \vdash t : A[X := F]}\ (\forall E)$$

$$\frac{\Gamma \vdash r : A}{\Gamma \vdash \mathsf{inl}\, r : A + B}\ (+I_L) \qquad \frac{\Gamma \vdash r : B}{\Gamma \vdash \mathsf{inr}\, r : A + B}\ (+I_R)$$

$$\frac{\Gamma \vdash r : A + B \quad \Gamma, x : A \vdash s : C \quad \Gamma, y : B \vdash t : C}{\Gamma \vdash \mathsf{case}(r, x.s, y.t) : C} \ (+E)$$

$$\frac{\Gamma \vdash r : A \quad \Gamma \vdash s : B}{\Gamma \vdash \langle r, s\rangle : A \times B} \ (\times I) \qquad \frac{\Gamma \vdash s : A \times B}{\Gamma \vdash \mathsf{fst}\, s : A} \ (\times E_L) \qquad \frac{\Gamma \vdash s : A \times B}{\Gamma \vdash \mathsf{snd}\, s : B} \ (\times E_R).$$

- Reduction. The operational semantics is given by the one-step $\beta$-reduction relation $t \to t'$ defined as the contextual closure of the following axioms:

$$\begin{array}{rcl} (\lambda x.r)s & \mapsto_\beta & r[x := s] \\ \mathsf{case}(\mathsf{inl}\, r, x.s, y.t) & \mapsto_\beta & s[x := r] \\ \mathsf{case}(\mathsf{inr}\, r, x.s, y.t) & \mapsto_\beta & t[y := r] \\ \mathsf{fst}\langle r, s\rangle & \mapsto_\beta & r \\ \mathsf{snd}\langle r, s\rangle & \mapsto_\beta & s. \end{array}$$

### 2.2. Algebras and coalgebras

We assume some knowledge of category theory, here we only state the basic concepts needed later, for full details on category theory see for example [18].

We will use the categorical approach to (co)induction (see [15]) to formulate our systems of (co)inductive types. This can be briefly stated as follows:

- Induction is the use of initiality for algebras.
- Coinduction is the use of finality for coalgebras.

Datatypes like natural numbers, lists or trees are modeled my initial algebras, whereas final coalgebras model codatatypes like streams, colists or infinite trees.

For our purpose we fix a default category $\mathcal{C}$ with finite products $\times$, initial object 1, finite coproducts $+$ and final object 0 such that products distribute over coproducts. An example of such category is *Set*.

**Definition 2.1.** Let $T : \mathcal{C} \to \mathcal{C}$ be a functor. A $T$-algebra is a pair $\langle A, f\rangle$ such that $f : TA \to A$. Analogously a $T$-coalgebra is a pair $\langle B, g\rangle$ with $g : B \to TB$.

Algebras and coalgebras form categories where morphisms are defined as follows.

**Definition 2.2.** Given two $T$-algebras $\langle A, f\rangle, \langle B, g\rangle$ a morphism from $\langle A, f\rangle$ to $\langle B, g\rangle$ is a $\mathcal{C}$-morphism $h : A \to B$ such that the following diagram commutes:

$$\begin{array}{ccc} TA & \xrightarrow{f} & A \\ {\scriptstyle Th}\downarrow & & \downarrow{\scriptstyle h} \\ TB & \xrightarrow{g} & B \end{array}$$

We say that the algebra $\langle A, f\rangle$ is initial if it is the initial object of the category of $T$-algebras, *i.e.*, if for every given algebra $\langle B, g\rangle$ there is a unique $h$ such that the above diagram commutes. In this case the $h$ is denoted $\mathsf{It}_g$ and called the iteratively defined morphism with step function $g$.

If the initial $T$-algebra exists, it is unique and is denoted as $\langle \mu T, \mathsf{in}_T\rangle$, so that $\mathsf{It}_g : \mu T \to B$ and

$$\mathsf{It}_g \circ \mathsf{in}_T = g \circ T(\mathsf{It}_g) \tag{2.1}$$

this equation is called principle of iteration.

Dually a morphism of coalgebras from $\langle B, g\rangle$ to $\langle A, f\rangle$ is a $\mathcal{C}$-morphism $h : B \to A$ such that the following diagram commutes:

$$\begin{array}{ccc} A & \xrightarrow{\ f\ } & TA \\ {\scriptstyle h}\uparrow & & \uparrow{\scriptstyle Th} \\ B & \xrightarrow{\ g\ } & TB \end{array}$$

We say that the coalgebra $\langle A, f\rangle$ is final if it is the final object of the category of $T$-algebras, *i.e.*, if for every given coalgebra $\langle B, g\rangle$ there is a unique $h$ such that the above diagram commutes. In this case we denote such $h$ with $\mathsf{CoIt}_g$ and call it the coiteratively defined morphism with step function $g$.

If the final $T$-coalgebra exists, it is unique and is denoted with $\langle \nu T, \mathsf{out}_T\rangle$, so that $\mathsf{CoIt}_g : B \to \nu T$ and

$$\mathsf{out}_T \circ \mathsf{CoIt}_g = F(\mathsf{CoIt}_g) \circ g \tag{2.2}$$

this equation is called principle of coiteration.

The existence of initial algebras and final coalgebras is guaranteed for a wide class of functors including all so-called polynomial functors built up from the identity and constant functors using products and coproducts. Let us recall some typical examples of (co)datatypes defined categorically.

**Example 2.1.** The natural numbers are defined as the initial algebra $\mathsf{Nat} = \mu T$ of the functor $T\,X = 1 + X$. In this case $\mathsf{in}_T : 1 + \mathsf{Nat} \to \mathsf{Nat}$ encodes the usual constructors zero and suc by means of $\mathrm{zero} = \mathsf{in}_T \circ \mathsf{inl}$ and $suc = \mathsf{in}_T \circ \mathsf{inr}$ where $\mathsf{inl}, \mathsf{inr}$ are the coproduct injections. As usual, in category theory, zero is a global element $zero : 1 \to \mathsf{Nat}$ and the element 0 is defined by $0 = \mathrm{zero} \star$. Moreover, given functions $c : 1 \to C$ and $g : C \to C$ the iteration morphism $f = \mathsf{It}_{[c,g]}$, where $[c, g] : 1 + C \to C$ is the usual copair of arrows, generates the following version of the principle of iteration: $f \circ \mathrm{zero} = c,\ f \circ \mathrm{suc} = g \circ f$, which is the usual iteration on naturals.

**Example 2.2.** The data type of finite lists over a given type $A$ is defined as the initial algebra $\mathsf{List}\,A = \mu T$ of the functor $T\,X = 1 + A \times X$. In this case

$\mathsf{in}_T : 1 + A \times \mathsf{List}\, A \to \mathsf{List}\, A$ encodes the usual constructors nil and cons by means of $nil = \mathsf{in}_T(\mathsf{inl}\star)$ and $cons = \mathsf{in}_T \circ \mathsf{inr}$. Given functions $c : 1 \to C$ and $g : A \times C \to C$ the iterative morphism $f = \mathsf{It}_{[c,g]}$ gives us the following version of the principle of iteration: $f(\mathrm{nil}) = c,\ f(\mathrm{cons}\langle x, xs\rangle) = g(\langle x, f(xs)\rangle)$, which is the usual iteration on lists and corresponds to the usual operator `foldr` in HASKELL.

**Example 2.3.** The codatatype of streams or strictly infinite lists over a given type $A$ is defined as the final coalgebra $\mathsf{Stream}\, A = \nu T$ of the functor $T\, X = A \times X$. The $\mathsf{out} : \mathsf{Stream}\, A \to A \times \mathsf{Stream}\, A$ arrow encodes the usual destructors *head* and *tail* defined by $\mathrm{head} = \mathsf{fst} \circ \mathsf{out},\ \mathrm{tail} = \mathsf{snd} \circ \mathsf{out}$. Moreover, given two morphisms $h : B \to A,\ t : B \to B$ the coiterative morphism $f = \mathsf{Colt}_{\langle h,t\rangle}$, where $\langle h, t\rangle : B \to A \times B$ is the usual pair of arrows, generates the following version of the coiteration principle: $head \circ f = h,\ tail \circ f = f \circ t$ which is essentially the unfold operator of [8].

**Example 2.4.** The final coalgebra $\mathsf{Colist}\, C = \nu T$, where $T\, X = C \times (1 + X)$, generates the codatatype of non-empty and maybe infinite lists of elements of $C$. In this case we have $tail : \mathsf{Colist}\, C \to 1 + \mathsf{Colist}\, C, tail = \mathsf{snd} \circ \mathsf{out}$. Therefore this destructor can return an error (the inhabitant of 1) indicating that the tail does not exist, in this case the colist is finite.

The following proposition, due to Lambek, states that the morphisms $\mathsf{in}, \mathsf{out}$ are isomorphisms.

**Proposition 2.1.** *$\mathsf{in}_T, \mathsf{out}_T$ are isomorphisms, therefore there exist inverse morphisms ${\mathsf{in}_T}^{-1}, {\mathsf{out}_T}^{-1}$ such that ${\mathsf{in}_T}^{-1} \circ \mathsf{in}_T = \mathsf{Id}_{T(\mu T)}$ and $\mathsf{out}_T \circ {\mathsf{out}_T}^{-1} = \mathsf{Id}_{T(\nu T)}$. These equations are called the principle of inductive and coinductive inversion respectively.*

*Proof.* Straightforward. □

The morphisms ${\mathsf{in}_T}^{-1}, {\mathsf{out}_T}^{-1}$ are useful to define inductive destructors and coinductive destructors.

**Example 2.5.** For the previous examples the inverse morphisms and the inversion principles behave as follows:

- Natural numbers: ${\mathsf{in}_T}^{-1} : \mathsf{Nat} \to 1 + \mathsf{Nat}$ such that ${\mathsf{in}_T}^{-1}(0) = \mathsf{inl}\star$, and ${\mathsf{in}_T}^{-1}(\mathsf{suc}\, n) = \mathsf{inr}\, n$ which corresponds to the predecessor function with error handling.
- Finite lists: ${\mathsf{in}_T}^{-1} : \mathsf{List}\, A \to 1 + A \times \mathsf{List}\, A$ such that ${\mathsf{in}_T}^{-1}(\mathsf{nil}) = \mathsf{inl}\star$, which corresponds to an error, and ${\mathsf{in}_T}^{-1}(\mathsf{cons}\langle x, xs\rangle) = \mathsf{inr}\langle x, xs\rangle$. This behavior allows us to define the pair of destructors $\langle \mathsf{head}, \mathsf{tail}\rangle$ by means of a case analysis.
- Streams: the morphism ${\mathsf{out}_T}^{-1} : A \times \mathsf{Stream}\, A \to \mathsf{Stream}\, A$ such that $\mathsf{out}({\mathsf{out}_T}^{-1}\langle a, s\rangle) = \langle a, s\rangle$ which allows a direct definition of the $\mathsf{cons}$ constructor of streams.

- Colists: the morphism $\mathsf{out}_T{}^{-1} : C \times (1 + \mathsf{Colist}\, C) \to \mathsf{Colist}\, C$ directly defines the $\mathsf{cons}$ constructor of colists. For instance $\mathsf{out}_T{}^{-1}\, \langle c, \mathsf{inl}\, \star \rangle$ represents the unitary list with element $c$.

Next, we recall the concepts of (co)recursive algebras, introduced in [7], whose universal properties will generate the so-called primitive (co)recursion principles.

**Definition 2.3.** Define $\Pi_D : \mathcal{C} \to \mathcal{C}$ as $\Pi_D C = C \times D$. We say that the $T$-algebra $\langle A, f \rangle$ is recursive if for every $T\Pi_A$-algebra $\langle B, g \rangle$ there exists a morphism $h : A \to B$ such that:

$$\begin{array}{ccc} TA & \xrightarrow{f} & A \\ {\scriptstyle T\langle \mathsf{Id}, h \rangle}\downarrow & & \downarrow{\scriptstyle h} \\ T(A \times B) & \xrightarrow{g} & B \end{array} \tag{2.3}$$

Set $\Sigma_D : \mathcal{C} \to \mathcal{C}$ with $\Sigma_D C = C + D$. We say that the $T$-coalgebra $\langle A, f \rangle$ is corecursive if for every $T\Sigma_A$-coalgebra $\langle B, g \rangle$ there exists a morphism $h : B \to A$ such that:

$$\begin{array}{ccc} A & \xrightarrow{f} & TA \\ {\scriptstyle h}\uparrow & & \uparrow{\scriptstyle T[\mathsf{Id}, h]} \\ B & \xrightarrow{g} & T(A + B) \end{array} \tag{2.4}$$

**Proposition 2.2.** *$\langle \mu T, \mathsf{in}_T \rangle$ is recursive and $\langle \nu T, \mathsf{out}_T \rangle$ is corecursive.*

*Proof.* Let $\langle B, g \rangle$ be a $T\Pi_{\mu T}$-algebra, *i.e.* $g : T(\mu T \times B) \to B$. It is easy to see that $\mathsf{in}_T \circ T(\mathsf{fst}) : T(\mu T \times B) \to \mu T$, so that we get the following $T$-algebra:

$$\langle \mathsf{in}_T \circ T(\mathsf{fst}), g \rangle : T(\mu T \times B) \to \mu T \times B.$$

Therefore by iteration there is a unique $h : \mu T \to \mu T \times B$ such that

$$h \circ \mathsf{in}_T = \big\langle \mathsf{in}_T \circ T(\mathsf{fst}), g \big\rangle \circ T(h). \tag{2.5}$$

The goal is to show that for the given $g$ there is an $h' : \mu T \to B$ such that

$$\begin{array}{ccc} T\mu T & \xrightarrow{\mathsf{in}_T} & \mu T \\ {\scriptstyle T(\langle \mathsf{id}, h' \rangle)}\downarrow & & \downarrow{\scriptstyle h'} \\ T(\mu T \times B) & \xrightarrow{g} & B \end{array} \tag{2.6}$$

Set $h' : \mu T \to B$ defined as $h' = \mathsf{snd} \circ h$, we will show that the diagram commutes, *i.e.*,

$$h' \circ \mathsf{in}_T = g \circ T(\langle \mathsf{Id}, h' \rangle).$$

First we show that $\mathsf{fst} \circ h = \mathsf{Id}$ by initiality, *i.e.* we have to show that the following diagram commutes:

$$\begin{array}{ccc} T\mu T & \xrightarrow{\mathsf{in}_T} & \mu T \\ {\scriptstyle T(\mathsf{fst} \circ h)}\downarrow & & \downarrow{\scriptstyle \mathsf{fst} \circ h} \\ T\mu T & \xrightarrow{\mathsf{in}_T} & \mu T \end{array}$$

we have by equation (2.5)

$$\begin{aligned} (\mathsf{fst} \circ h) \circ \mathsf{in}_T = \mathsf{fst} \circ (h \circ \mathsf{in}_T) &= \mathsf{fst} \circ \Big( \langle \mathsf{in}_T \circ T(\mathsf{fst}), g \rangle \circ T(h) \Big) \\ &= \big( \mathsf{fst} \circ \langle \mathsf{in}_T \circ T(\mathsf{fst}), g \rangle \big) \circ T(h) \\ &= \big( \mathsf{in}_T \circ T(\mathsf{fst}) \big) \circ T(h) = \mathsf{in}_T \circ T(\mathsf{fst} \circ h). \end{aligned}$$

Therefore the diagram commutes, and by uniqueness we have $\mathsf{fst} \circ h = \mathsf{Id}$.

Next observe that $h = \langle \mathsf{fst} \circ h, \mathsf{snd} \circ h \rangle = \langle \mathsf{Id}, h' \rangle$. Now we can show that diagram (2.2) commutes:

$$\begin{aligned} h' \circ \mathsf{in}_T &= \big( \mathsf{snd} \circ h \big) \circ \mathsf{in}_T \\ &= \mathsf{snd} \circ \big( h \circ \mathsf{in}_T \big) \\ &= \mathsf{snd} \circ \Big( \langle \mathsf{in}_T \circ T(\mathsf{fst}), g \rangle \circ T(h) \Big) \\ &= \Big( \mathsf{snd} \circ \langle \mathsf{in}_T \circ T(\mathsf{fst}), g \rangle \Big) \circ T(h) \\ &= g \circ T(h) \\ &= g \circ T(\langle \mathsf{Id}, h' \rangle). \end{aligned}$$

Therefore diagram (2.2) commutes.

The case for the final coalgebra is similar. □

For the cases of the initial algebra and the final coalgebra, the $h$ that makes diagrams (2.3), (2.4) commute is denoted $\mathsf{Rec}_g, \mathsf{CoRec}_g$ respectively and we refer to them as the (co)recursively defined morphism with step function $g$, so that we have $\mathsf{Rec}_g : \mu T \to B,\ \mathsf{CoRec}_g : B \to \nu T$ such that the following principles hold:

- Principle of primitive recursion

$$\mathsf{Rec}_g \circ \mathsf{in}_T = g \circ T(\langle \mathsf{Id}, \mathsf{Rec}_g \rangle). \tag{2.7}$$

- Principle of primitive corecursion

$$\mathsf{out}_T \circ \mathsf{CoRec}_g = T([\mathsf{Id}, \mathsf{CoRec}_g]) \circ g. \tag{2.8}$$

Let us interpret these principles in some particular cases.

**Example 2.6.** For natural numbers, finite lists and streams the (co)recursion principle behaves as follows:

- Natural numbers: given functions $c : 1 \to C$ and $g : \mathsf{Nat} \times C \to C$ the recursive morphism $f = \mathsf{Rec}_{[c,g]}$, generates the following version of the principle of recursion: $f(0) = c,\ f\,(\mathsf{suc}\, n) = g\langle n, f(n)\rangle$, which is the usual primitive recursion principle on naturals.
- Finite lists: given functions $c : 1 \to C$ and $g : A \times ((\mathsf{List}\, A) \times C) \to C$ the recursive morphism $f = \mathsf{Rec}_{[c,g]}$, generates the following version of the principle of recursion: $f(\mathsf{nil}) = c,\ f\,(\mathsf{cons}\langle x, xs\rangle) = g\langle x, \langle xs, f\, xs\rangle\rangle$.
- Streams: given two morphisms $h : B \to A,\ t : B \to (\mathsf{Stream}\, A) + B$ the corecursive morphism $f = \mathsf{CoIt}_{\langle h,t\rangle}$, generates the following version of the corecursion principle: $\mathsf{head}(f\, x) = h\, x,\ \mathsf{tail}(f\, x) = \mathsf{case}\ (t\, x)\ \mathsf{of}\ \mathsf{Inl}\, y \Rightarrow y \mid \mathsf{Inr}\, z \Rightarrow f\, z$.

## 2.3. Dialgebras

The concept of dialgebra, introduced in [14], is a straightforward generalization of (co)algebras with stronger expressive power (see [30]). With dialgebras we can represent products, coproducts and even exponential objects (see [6]). We will serve later from this concept to justify the clausular feature of one type system.

**Definition 2.4.** Let $F, G : \mathcal{C} \to \mathcal{D}$ be covariant functors between two categories $\mathcal{C}, \mathcal{D}$. A $F, G$-dialgebra is a pair $\langle A, f\rangle$ where $A$ is a $\mathcal{C}$-object and $f : FA \to GA$ is a $\mathcal{D}$-morphism.

**Definition 2.5.** A morphism between two $F, G$-dialgebras $\langle A, f\rangle, \langle B, g\rangle$ is a $\mathcal{C}$-morphism $h : A \to B$ such that:

$$\begin{array}{ccc} FA & \xrightarrow{f} & GA \\ {\scriptstyle Fh}\downarrow & & \downarrow{\scriptstyle Gh} \\ FB & \xrightarrow{g} & GB \end{array}$$

Observe that if $I$ is the identity functor then a $T, I$-dialgebra $\langle A, f\rangle$ is a $T$-algebra and an $I, T$-dialgebra is a $T$-coalgebra.

For our later purposes we are only interested in initial and final dialgebras involving the functors $F, \mathbb{I} : \mathcal{C} \to \mathcal{C}^k$ defined by $F = \langle F_1, \ldots, F_k\rangle$, $\mathbb{I} = \langle I, \ldots, I\rangle$ where $F_i : \mathcal{C} \to \mathcal{C}$ are arbitrary functors and $I : \mathcal{C} \to C$ is again the identity functor.

For variety, we first discuss finality for this kind of functors. If the final $\mathbb{I}, F$-dialgebra exists, it will be denoted with $\langle \nu(F_1, \ldots, F_k), \mathsf{out}_k\rangle$. The finality property is given by the following diagram, where $V = \nu(F_1, \ldots, F_k)$

$$\begin{array}{ccc} \langle V, \ldots, V\rangle & \xrightarrow{\mathsf{out}_k} & \langle F_1 V, \ldots, F_k V\rangle \\ {\scriptstyle \langle h, \ldots, h\rangle}\uparrow & & \uparrow{\scriptstyle \langle F_1 h, \ldots, F_k h\rangle} \\ \langle B, \ldots, B\rangle & \xrightarrow{g} & \langle F_1 B, \ldots, F_k B\rangle \end{array}$$

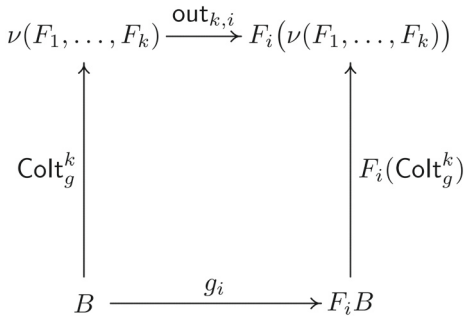
where $h : B \to V$ is the unique function such that:

$$\mathsf{out}_k \circ \langle h, \ldots, h\rangle = \langle F_1 h, \ldots, F_k h\rangle \circ g.$$

Observe that the morphisms $\mathsf{out}_k$ and $g$ are necessarily of the form

$$\mathsf{out}_k = \langle \mathsf{out}_{k,1}, \ldots, \mathsf{out}_{k,k}\rangle \qquad g = \langle g_1, \ldots, g_k\rangle.$$

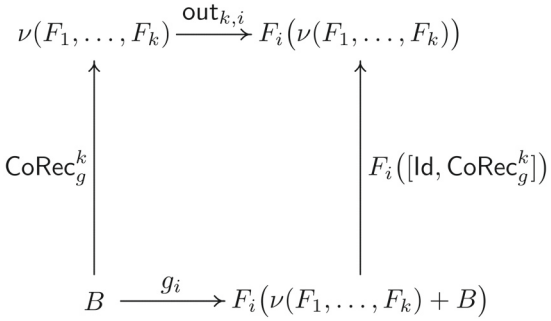
Therefore the previous diagram can be splitted into the following $k$ diagrams, where we denote the unique $h$ above with $\mathsf{Colt}^k_g$.




$$\mathsf{out}_{k,i} \circ \mathsf{Colt}_g^k = F_i(\mathsf{Colt}_g^k) \circ g_i. \tag{2.9}$$

These equations represent the coiteration principle on dialgebras.

Analogously, corecursion is introduced by the following $k$-diagrams:




$$\mathsf{out}_{k,i} \circ \mathsf{CoRec}_g^k = F_i\big([\mathsf{Id},\mathsf{CoRec}_g^k]\big) \circ g_i. \tag{2.10}$$

These equations represent the principle of primitive corecursion on dialgebras.

Finally the coinductive inversion principle provides us with a morphism $\mathsf{out}_k^{-1} : \langle F_1V,\dots,F_kV\rangle \to \langle V,\dots,V\rangle$ such that $\mathsf{out}_k \circ \mathsf{out}_k^{-1} = I_{\langle F_1V,\dots,F_kV\rangle}$. However, in practice, we will restrict the final dialgebra morphism to $\mathsf{out}_k : V \to \langle F_1V,\dots,F_kV\rangle$, therefore getting an inverse $\mathsf{out}_k^{-1} : \langle F_1V,\dots,F_kV\rangle \to V$ such that the following $k$ equations hold:

$$\mathsf{out}_{k,i} \circ \mathsf{out}_k^{-1} = \pi_i \tag{2.11}$$

where $\pi_i : \langle F_1V,\dots,F_kV\rangle \to F_iV$ is the usual projection.

Let us see the definition of streams as a final dialgebra.

**Example 2.7.** The codatatype of streams over a given type $A$ is defined as the final dialgebra $\mathsf{Stream}\,A = \nu(F_1,F_2)$ of the functors $F_1\,X = A,\ F_2\,X = X$. The usual destructors $\mathsf{head} : \mathsf{Stream}\,A \to A$ and $\mathsf{tail} : \mathsf{Stream}\,A \to \mathsf{Stream}\,A$ are directly defined as $\mathsf{head} = \mathsf{out}_{2,1},\ \mathsf{tail} = \mathsf{out}_{2,2}$. Moreover, given two morphisms $g_1 : B \to A,\ g_2 : B \to B$ the coiterative morphism $f = \mathsf{Colt}_g^2$, where $g = \langle g_1, g_2\rangle$,

generates the following version of the coiteration principle: $\mathsf{head} \circ f = g_1$, $\mathsf{tail} \circ f = f \circ g_2$. Observe that the need for projections disappeared.

Dually, and denoting with $\langle \mu(F_1, \ldots, F_k), \mathsf{in}_k \rangle$ the initial $F, \mathbb{I}$-dialgebra we arrive to the following $k$ diagrams, where $\mathsf{in}_k = \langle \mathsf{in}_{k,1}, \ldots, \mathsf{in}_{k,k} \rangle$ and $g = \langle g_1, \ldots, g_k \rangle$.

$$\begin{array}{ccc} F_i\big(\mu(F_1, \ldots, F_k)\big) & \xrightarrow{\mathsf{in}_{k,i}} & \mu(F_1, \ldots, F_k) \\ \Big\downarrow{\scriptstyle F_i(\mathsf{It}_g^k)} & & \Big\downarrow{\scriptstyle \mathsf{It}_g^k} \\ F_i B & \xrightarrow{\quad g_i \quad} & B \end{array}$$

Therefore the iteration principle is given by the following $k$-equations:

$$\mathsf{It}_g^k \circ \mathsf{in}_{k,i} = g_i \circ F_i(\mathsf{It}_g^k). \tag{2.12}$$

With respect to primitive recursion we get the following $k$ diagrams and equations:

$$\begin{array}{ccc} F_i\big(\mu(F_1, \ldots, F_k)\big) & \xrightarrow{\mathsf{in}_{k,i}} & \mu(F_1, \ldots, F_k) \\ \Big\downarrow{\scriptstyle F_i\big(\langle \mathsf{Id}, \mathsf{Rec}_g^k \rangle\big)} & & \Big\downarrow{\scriptstyle \mathsf{Rec}_g^k} \\ F_i\big(\mu(F_1, \ldots, F_k) \times B\big) & \xrightarrow{\quad g_i \quad} & B \end{array}$$

$$\mathsf{Rec}_g^k \circ \mathsf{in}_{k,i} = g_i \circ F_i\big(\langle \mathsf{Id}, \mathsf{Rec}_g^k \rangle\big). \tag{2.13}$$

Finally the inductive inversion principle provides us with a morphism $\mathsf{in}_k^{-1} : \langle \mu, \ldots, \mu \rangle \to \langle F_1 \mu, \ldots, F_k \mu \rangle$ such that

$$\mathsf{in}_k^{-1} \circ \mathsf{in}_k = I_{\langle F_1 \mu, \ldots, F_k \mu \rangle} \tag{2.14}$$

where $\mu = \mu(F_1, \ldots, F_k)$.

Let us see the definition of natural numbers as an initial dialgebra.

**Example 2.8.** The natural numbers are defined as the initial dialgebra $\mathsf{Nat} = \mu(F_1, F_2)$ where $F_1\, X = 1$ and $F_2\, X = X$. In this case we have $\mathsf{in}_2 = \langle \mathsf{in}_{2,1}, \mathsf{in}_{2,2} \rangle$

where $\mathsf{in}_{2,1} : 1 \to \mathsf{Nat}$ corresponds to a global *zero* and $\mathsf{in}_{2,2} : \mathsf{Nat} \to \mathsf{Nat}$ plays the role of the successor function *suc*. Moreover, given a pair of functions $g = \langle g_1, g_2 \rangle$ where $g_1 : 1 \to C$ and $g_2 : C \to C$ the iteration morphism $f = \mathsf{It}_g^2$, generates the following version of the principle of iteration: $f \circ \mathrm{zero} = g_1$, $f \circ \mathrm{suc} = g_2 \circ f$, which is the usual iteration on naturals. Observe that the use of injections disappeared.

Next, we discuss how to model the above categorical schemes of recursion and inversion in the framework of type systems.

### 2.4. From categories to types

Our goal is to implement the principles of (co)iteration, primitive (co)recursion and (co)inductive inversion as defined above, by means of type and term constructors extending system $\mathsf{F}$. To implement these categorical schemes as constructors of typed lambda calculi, we need to see the type system as a category $\mathcal{T}$ where types are objects, morphisms are transformations between types, and composition is function composition. Such categories of types and their features are well-known, see for example [3].

A functor $F : \mathcal{T} \to \mathcal{T}$ is then a transformation between types. In particular we use functors $F(X)$ depending on a type variable $X$, which map a type $B$ to a type $F(B)$. Such functors are defined, more accurately, by expressions of the form $\lambda X.F$ abstracting the type variable $X$. Note that the systems developed in this paper are not higher-order and therefore abstractions like $\lambda X.F$ are only a useful notation. In particular an application $(\lambda X.F)B$ of a functor to a type $B$ will always be identified with the capture-avoiding substitution $F[X := B]$. It is important to mention that for a type transformer $F$ to qualify as a functor is necessary to define its action on morphisms. This is necessary to model the categorical combinators. For instance, the iteration principle $\mathsf{It}_g \circ \mathsf{in}_F = g \circ F(\mathsf{It}_g)$ involves the application of the functor $F$ to a morphism $\mathsf{It}_g$. Hence, as in our type theoretical framework $F$ is a type transformer and $\mathsf{It}_g$ is not a type but a functional term, we need to give a definition to applications like $F(\mathsf{It}_g)$. To guarantee the functoriality or monotonicity of $\lambda X.F$ in functional terms, a syntactical condition on the type variable $X$ is usually required, namely that $X$ must occur only on positive positions in $F$, that is, not to the left of an odd number of the $\to$ type constructor (see, for example [7,12,13,40]). In our treatment we prefer to follow [20] and use full monotonicity instead: the functoriality of $\lambda X.F$ in functional terms is represented internally by means of a term $\mathsf{map} : F\ \mathsf{mon}\ X$ in a given context. The type $F\ \mathsf{mon}\ X$, defined as $\forall X \forall Y.(X \to Y) \to F \to F[X := Y]$, represents the fact that the functor $\lambda X.F$ is monotone (covariant) with respect to its argument $X$. Such terms are called *monotonicity witnesses*. For the merits of using full-monotonicity instead of positivity we point to [1]. Moreover, it is important to remark that a positive system is not compatible with Curry-style systems in the sense discussed in Section 3.3. In conclusion, a functor in our framework is a pair $\langle \lambda X.F, \mathsf{map} \rangle$ where $\mathsf{map}$ is a term of type $F\ \mathsf{mon}\ X$ in the needed context. Thus, our functors are just like instances of the class `Functor` in Haskell. An example is in order.

**Example 2.9.** Consider the type transformer $\lambda X.F$, with $F = 1 + A \times X$ and $A$ a constant type. For this transformation to be a functor we need to define a monotonicity witness of type $(1 + A \times X)\,\mathsf{mon}\,X$, that is a term $\mathsf{map}$ of type $\forall X \forall Y.(X \to Y) \to (1 + A \times X) \to (1 + A \times Y)$. Given $f : X \to Y$ and $t : 1 + A \times X$ the natural idea to get an inhabitant of $1 + A \times Y$ is to perform a case analysis on $t$: if $t = \mathsf{inl}\,\star$ we just return $t$ and if $t = \mathsf{inr}\,\langle a, s\rangle$ then $\mathsf{inr}\,\langle a, f\,s\rangle$ is returned. This leads us to define $\mathsf{map} = \lambda f.\lambda x.\mathsf{case}(x, y.\,\mathsf{inl}\,y, z.\,\mathsf{inr}\langle \mathsf{fst}\,z, f\,(\mathsf{snd}\,z)\rangle)$.

Once we have settled what a functor is in a type-theoretical setting, we proceed to define the categorical recursion, corecursion and inversion principles discussed in Sections 2.2 and 2.3.

## 3. A type system for (co)algebras

Following the above ideas we define next an extension of system F modeling the categorical principles discussed in Section 2.2. For this purpose, given a functor $\lambda X.F$ we will allow the construction of a so-called inductive type $\mu X.F$ modeling the initial algebra of $\lambda X.F$ or the construction of a so-called coinductive type $\nu X.F$ representing the final coalgebra. Moreover we will introduce term constructors for (co)iteration, primitive (co)recursion and inversion.

### 3.1. Definition of the system

We extend system F as follows:

- Types:
$$A, B, C, F, G ::= \ldots \mid \mu X.F \mid \nu X.F.$$
Observe that the formation of (co)inductive types does not put a restriction on $F$, the functoriality requirement will be ensured by the typing rules defined below.
- Terms: we add a term constructor for each categorical operation described in Section 2.2.
$$\begin{aligned} t, r, s, m \quad ::= \ldots \quad & \mid \mathsf{It}(m, s, t) \mid \mathsf{Rec}(m, s, t) \mid \mathsf{in}\,t \mid \mathsf{in}^{-1}(m, t) \mid \\ & \mathsf{CoIt}(m, s, t) \mid \mathsf{CoRec}(m, s, t) \mid \mathsf{out}\,t \mid \mathsf{out}^{-1}(m, t). \end{aligned}$$
- Typing rules: we add a typing rule for every new term constructor as follows:
  - Introduction of inductive types:
$$\frac{\Gamma \vdash t : F[X := \mu X.F]}{\Gamma \vdash \mathsf{in}\,t : \mu X.F}\ (\mu I).$$

– Elimination of inductive types:
∗ By inversion:

$$\frac{\Gamma \vdash t : \mu X.F \quad \Gamma \vdash m : F\,\mathsf{mon}\,X}{\Gamma \vdash \mathsf{in}^{-1}(m,t) : F[X := \mu X.F]}\ (\mu E^i).$$

∗ By iteration:

$$\frac{\Gamma \vdash t : \mu X.F \quad \Gamma \vdash m : F\,\mathsf{mon}\,X \quad \Gamma \vdash s : F[X := B] \to B}{\Gamma \vdash \mathsf{It}(m,s,t) : B}\ (\mu E).$$

∗ By primitive recursion:

$$\frac{\Gamma \vdash t : \mu X.F \quad \Gamma \vdash m : F\,\mathsf{mon}\,X \quad \Gamma \vdash s : F[X := (\mu X.F) \times B] \to B}{\Gamma \vdash \mathsf{Rec}(m,s,t) : B}\ (\mu E^+).$$

– Introduction of coinductive types:
∗ By coiteration:

$$\frac{\Gamma \vdash s : B \to F[X := B] \quad \Gamma \vdash m : F\,\mathsf{mon}\,X \quad \Gamma \vdash t : B}{\Gamma \vdash \mathsf{CoIt}(m,s,t) : \nu X.F}\ (\nu I).$$

∗ By primitive corecursion:

$$\frac{\Gamma \vdash s : B \to F[X := (\nu X.F) + B] \quad \Gamma \vdash m : F\,\mathsf{mon}\,X \quad \Gamma \vdash t : B}{\Gamma \vdash \mathsf{CoRec}(m,s,t) : \nu X.F}\ (\nu I^+).$$

∗ By inversion:

$$\frac{\Gamma \vdash t : F[X := \nu X.F] \quad \Gamma \vdash m : F\,\mathsf{mon}\,X}{\Gamma \vdash \mathsf{out}^{-1}(m,t) : \nu X.F}\ (\nu I^i).$$

– Elimination of coinductive types:

$$\frac{\Gamma \vdash r : \nu X.F}{\Gamma \vdash \mathsf{out}\,r : F[X := \nu X.F]}\ (\nu E).$$

- Operational semantics: it is given by extending the one-step $\beta$-reduction relation $t \to_\beta t'$ with the following axioms under contextual closure.

$$\begin{array}{rcl}
\mathsf{It}(m, s, \mathsf{in}\, t) & \mapsto_\beta & s\Big(m\big(\lambda x.\mathsf{It}(m, s, x)\big)t\Big) \\
\mathsf{Rec}(m, s, \mathsf{in}\, t) & \mapsto_\beta & s\Big(m\Big(\big\langle \mathsf{Id}, \lambda z.\mathsf{Rec}(m, s, z)\big\rangle\Big)t\Big) \\
\mathsf{in}^{-1}(m, \mathsf{in}\, t) & \mapsto_\beta & m(\lambda z.z)t \\
\mathsf{out}\,\mathsf{CoIt}(m, s, t) & \mapsto_\beta & m\big(\lambda z.\mathsf{CoIt}(m, s, z)\big)(st) \\
\mathsf{out}\,\mathsf{CoRec}(m, s, t) & \mapsto_\beta & m\big([\mathsf{Id}, \lambda x.\mathsf{CoRec}(m, s, x)]\big)(st) \\
\mathsf{out}\,\mathsf{out}^{-1}(m, t) & \mapsto_\beta & m(\lambda z z)t
\end{array}$$

where $\mathsf{Id} = \lambda x.x$ and for given $f : A \to B, g : C \to B$ we define the copair operator on functions $[f, g] : A + C \to B$ as $[f, g] = \lambda z.\mathsf{case}(z, x.fx, y.gy)$. Analogously for $f : B \to A, g : B \to C$, the pair operator on functions $\langle f, g\rangle : B \to A \times C$ is defined as $\langle f, g\rangle = \lambda z.\langle fz, gz\rangle$.

As always the transitive closure of $\to_\beta$ is denoted by $\to_\beta^+$ and the reflexive-transitive closure by $\to_\beta^\star$.

It is worth noting that each of the above reduction rules originates in one of the categorical principles of (co)iteration, (co)recursion or inversion discussed in Section 2.2.

This finishes the definition of the type system MICT of *monotone inductive and coinductive types*.

### 3.1.1. *On inversion*

The reduction rules for inversion may look awkward at first sight and deserve some explanation. For instance, in the case of coinductive inversion, being $\mathsf{out}$ and $\mathsf{out}^{-1}$ inverses in the categorical setting, it may seem strange not to define the rule directly as $\mathsf{out}\,\mathsf{out}^{-1}(m, t) \mapsto_\beta t$. However, this naive rule is ruled out as it destructs the termination of the system which can easily be seen as follows:

Let 1 be the unit type with inhabitant $\star$ (see Ex. 3.1, p. 720). Define $\mathsf{T} = \nu X.X \to 1, m = \lambda f \lambda x \lambda y.\star$, $\omega = \lambda x.(\mathsf{out}\, x)x$, and $\Omega = \omega(\mathsf{out}^{-1}(m, \omega))$. We have the typings $\vdash m : (X \to 1)\,\mathsf{mon}\,X$, $\vdash \omega : \mathsf{T} \to 1, \vdash \mathsf{out}^{-1}(m, \omega) : \mathsf{T}$ and $\vdash \Omega : 1$. With the rule $\mathsf{out}\,\mathsf{out}^{-1}(m, t) \mapsto_\beta t$ we get $\Omega \to_\beta^+ \Omega$:

$$\Omega \to_\beta (\mathsf{out}\,\mathsf{out}^{-1}(m, \omega))(\mathsf{out}^{-1}(m, \omega)) \to_\beta \omega(\mathsf{out}^{-1}(m, \omega)) = \Omega.$$

Therefore we rule out such reduction. This phenomenon was originally noticed in [21] for fixed-point types.

## 3.2. Programming in MICT

In this section we develop several examples of (co)inductive types and programs in MICT. In each example we present an inductive type $\mu X.F$ or a coinductive type $\nu X.F$ together with a so-called canonical monotonicity witness $\mathsf{map} : F\,\mathsf{mon}\,X$, mechanically defined depending on the syntactical form of the type $F$ (see Appendix B). Our inductive examples model usual datatypes whose inhabitants are

finite structures built by the constructors encoded by the $\mathsf{in}$ term constructor. Dually, the coinductive examples represent codatatypes inhabited by strictly or potentially infinite structures; the emphasis being in their destructors, encoded by the $\mathsf{out}$ term constructor. We also present inductive destructors which are efficiently defined either by the inversion operator $\mathsf{in}^{-1}$ or by the primitive recursion operator $\mathsf{Rec}$, and coinductive constructors defined by the primitive corecursion operator $\mathsf{CoRec}$ or by the inversion operator $\mathsf{out}^{-1}$. Each example is finished with some functions involving the respective (co)datatype and implemented by one of the principles available in the system.

Before our series of examples let us explain the generalities of function programming in $\mathsf{MICT}$. Given an inductive type $\mu X.F$ we can program functions $g : (\mu X.F) \to B$ by the iteration or primitive recursion principles directly implemented in our operational semantics. If we define $g = \lambda z.\mathsf{It}(m, s, z)$, for some given monotonicity witness $m$ and step-function $s$, the specification by the principle of iteration

$$g(\mathsf{in}\, x) = s\big(m(g)(x)\big)$$

holds, in the sense that $g(\mathsf{in}\, x) \to_\beta^+ s\big(m(g)x\big)$.

Analogously primitive recursion provides a mean to program functions $g : (\mu X.F) \to B$ specified by

$$g(\mathsf{in}\, x) = s\big(m(\langle \mathsf{Id}, g\rangle)(x)\big).$$

Since this time we get $g(\mathsf{in}\, x) \to_\beta^+ s\big(m(\langle \mathsf{Id}, g\rangle)(x)\big)$, by defining $g = \lambda z.\mathsf{Rec}(m, s, z)$.

In a dual way, given a coinductive type $\nu X.F$ we can program functions $g : B \to \nu X.F$ as follows:

- By coiteration: if $g$ is specified by

  $$\mathsf{out}\, g(x) = m(g)(s(x))$$

  then define $g = \lambda x.\mathsf{CoIt}(m, s, x)$.
- By primitive corecursion: if $g$ is specified by

  $$\mathsf{out}\, g(x) = m([\mathsf{Id}, g])(s(x))$$

  then define $g = \lambda x.\mathsf{CoRec}(m, s, x)$.

In the following examples for any unary term constructor $\mathsf{c}$ we will denote the function $\lambda x.\mathsf{c}\, x$ simply with the same name $\mathsf{c}$. For example, the typing $\mathsf{in} : F[X := \mu X.F] \to \mu X.F$ must be understood as $\lambda x.\, \mathsf{in}\, x : F[X := \mu X.F] \to \mu X.F$. On the other hand, we sometimes use the anonymous variable $\_$ for dummy bindings, as in $\lambda \_.r$.

We start our series of examples with the very useful unit type 1 encoded directly in system $\mathsf{F}$.

**Example 3.1** (unit type)**.** The type 1, defined by $1 = \forall X.X \to X$, is characterized by having a unique element denoted $\star$. This type is mostly useful in the definition

of (co)inductive types with basic inhabitants, and as a way to handle errors. For instance, a sum type of the form $1+A$ has an error constant defined by $\mathsf{error} = \mathsf{inl}\,\star$. This error handling method will be present in some of the examples below. The reader can observe that $1 + A$ is essentially the type `maybe A` of HASKELL.

**Example 3.2** (Booleans)**.** This very simple but useful type can be defined as $\mathsf{Bool} = \mu X.1 + 1$

- Canonical monotonicity witness: $\mathsf{mapbool} = \lambda f \lambda x.x$.
- Constructors: $\mathsf{true} = \mathsf{in}(\mathsf{inl}\,\star)$, $\mathsf{false} = \mathsf{in}(\mathsf{inr}\,\star)$.
- The usual conditional $\mathsf{if\_then\_else\_} : \mathsf{Bool} \to A \to A \to A$ is defined by $\mathsf{if\ \_\ then\ \_\ else\ \_} = \lambda z \lambda x \lambda y.\mathsf{It}(\mathsf{mapbool}, s, z)$ where $s : 1 + 1 \to A \to A \to A$ is defined by $s = \lambda b.\lambda t \lambda f.\mathsf{case}(b, y.t, z.f)$.

The reader can observe that the binding in $\mu X.1 + 1$ is dummy and therefore the booleans can simply be defined as $\mathsf{Bool} = 1 + 1$.

**Example 3.3** (natural numbers)**.** Define $\mathsf{Nat} = \mu X.1 + X$ with $\mathsf{in} : 1 + \mathsf{Nat} \to \mathsf{Nat}$

- Canonical monotonicity witness: $\mathsf{mapnat} = \lambda f \lambda x.\mathsf{case}(x, u.\,\mathsf{inl}\,u, v.\,\mathsf{inr}(f\,v))$.
- Constructors:
  - Zero: $0 : \mathsf{Nat},\ 0 = \mathsf{in}(\mathsf{inl}\,\star)$.
  - Successor function: $\mathsf{suc} : \mathsf{Nat} \to \mathsf{Nat},\ \mathsf{suc} = \lambda n.\,\mathsf{in}(\mathsf{inr}\,n)$.
- Destructor: the destructor of naturals is the predecessor function $\mathsf{pred} : \mathsf{Nat} \to 1 + \mathsf{Nat}$, such that $\mathsf{pred}\ 0 = \mathsf{error},\ \mathsf{pred}(\mathsf{suc}\,n) = \mathsf{inr}\ n$. It can be defined either by primitive recursion or by inductive inversion.
  - Recursive predecessor:
    $\mathsf{pred} = \lambda n.\mathsf{Rec}(\mathsf{mapnat}, \lambda y.\mathsf{case}(y, u.\,\mathsf{inl}\,u, v.\,\mathsf{inr}(\mathsf{fst}\,v)), n)$.
  - Predecessor by inversion: $\mathsf{pred} = \lambda n.\,\mathsf{in}^{-1}(\mathsf{mapnat}, n)$. This shows that the presence of inductive inversion greatly simplifies the programming task.
- Some functions on $\mathsf{Nat}$:
  - $\mathsf{sum} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}, \mathsf{sum} = \lambda n \lambda z.\mathsf{It}(\mathsf{mapnat}, [\lambda\_.n, \mathsf{suc}], z)$.
  - $\mathsf{prod} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}, \mathsf{prod} = \lambda n \lambda z.\mathsf{It}(\mathsf{mapnat}, [\lambda\_.0, \lambda x.\mathsf{sum}\,x\,n], z)$.

**Example 3.4** (finite lists over $A$)**.** Define $\mathsf{List}\,A = \mu X.1 + A \times X$ with $\mathsf{in} : 1 + A \times \mathsf{List}\,A \to \mathsf{List}\,A$

- Canonical monotonicity witness:
  $\mathsf{maplist} = \lambda f \lambda x.\mathsf{case}(x, u.\,\mathsf{inl}\,u,\ v.\,\mathsf{inr}\langle \mathsf{fst}\,v, f\ \mathsf{snd}\,v\rangle)$.
- Constructors:
  - Empty list: $\mathsf{nil} : \mathsf{List}\,A,\quad \mathsf{nil} = \mathsf{in}(\mathsf{inl}\,\star)$.
  - Cons function: $\mathsf{cons} : A \times \mathsf{List}\,A \to \mathsf{List}\,A,\quad \mathsf{cons} = \lambda x.\,\mathsf{in}(\mathsf{inr}\,x)$.
- Destructors:
  - Recursive head function: $\mathsf{head} : \mathsf{List}\,A \to 1 + A$ such that $\mathsf{head}\,\mathsf{nil} = \mathsf{error},\ \mathsf{head}(\mathsf{cons}\langle a, \ell\rangle) = \mathsf{inr}\,a$
    $\mathsf{head} = \lambda z.\mathsf{Rec}(\mathsf{maplist}, \lambda y.\mathsf{case}(y, u.\,\mathsf{inl}\,u, v.\,\mathsf{inr}(\mathsf{fst}\,v)), z)$.
  - Recursive tail function: $\mathsf{tail} : \mathsf{List}\,A \to 1 + \mathsf{List}\,A$ such that $\mathsf{tail}\,\mathsf{nil} = \mathsf{error},\ \mathsf{tail}(\mathsf{cons}\langle a, \ell\rangle) = \mathsf{inr}\,\ell$
    $\mathsf{tail} = \lambda z.\mathsf{Rec}(\mathsf{maplist}, \lambda y.\mathsf{case}(y, u.\,\mathsf{inl}\,u, v.\,\mathsf{inr}(\mathsf{fst}(\mathsf{snd}\,v))), z)$.

  - Head by inversion: $\mathsf{head} = \lambda\ell.\mathsf{case}(\mathsf{in}^{-1}(\mathsf{maplist}, \ell), x.\,\mathsf{inl}\,x, y.\,\mathsf{inr}(\mathsf{fst}\,y))$.
  - Tail by inversion: $\mathsf{tail} = \lambda\ell.\mathsf{case}(\mathsf{in}^{-1}(\mathsf{maplist}, \ell), x.\,\mathsf{inl}\,x, y.\,\mathsf{inr}(\mathsf{snd}\,y))$.
- Some functions on $\mathsf{List}\,A$:
  - Append: $\mathsf{app} : \mathsf{List}\,A \to \mathsf{List}\,A \to \mathsf{List}\,A$,
    $\mathsf{app} = \lambda z.\mathsf{It}(\mathsf{maplist}, [\lambda y\lambda w.w,\ \lambda p\lambda x.\,\mathsf{cons}\langle\mathsf{fst}\,p, (\mathsf{snd}\,p)x\rangle], z)$.
  - Length: $\mathsf{length} : \mathsf{List}\,A \to \mathsf{Nat}$,
    $\mathsf{length} = \lambda z.\mathsf{It}(\mathsf{maplist}, [\lambda\_.0,\ \lambda p.\,\mathsf{suc}(\mathsf{snd}\,p)], z)$.
  - Reverse: $\mathsf{rev} : \mathsf{List}\,A \to \mathsf{List}\,A$,
    $\mathsf{rev} = \lambda z.\mathsf{It}\big(\mathsf{maplist}, [\lambda\_.\mathsf{nil},\ \lambda p.\mathsf{app}\,(\mathsf{snd}\,p)(\,\mathsf{cons}\langle\mathsf{fst}\,p, \mathsf{nil}\rangle)], z\big)$.

**Example 3.5** (unlabelled binary trees)**.** Define $\mathsf{BinTree} = \mu X.1 + X \times X$ with $\mathsf{in} : 1 + \mathsf{BinTree} \times \mathsf{BinTree} \to \mathsf{BinTree}$

- Canonical monotonicity witness:
  $\mathsf{mapbtree} = \lambda f\lambda x.\mathsf{case}(x, u.\,\mathsf{inl}\,u, v.\,\mathsf{inr}\langle f(\mathsf{fst}\,v),\ f(\mathsf{snd}\,v)\rangle)$.
- Constructors:
  - $\mathsf{node} : \mathsf{BinTree} \quad \mathsf{node} = \mathsf{in}(\mathsf{inl}\,\star)$.
  - $\mathsf{makebt} : \mathsf{BinTree} \times \mathsf{BinTree} \to \mathsf{BinTree},\ \mathsf{makebt} = \lambda x.\,\mathsf{in}(\mathsf{inr}\,x)$.
- Destructor: $\mathsf{strees} : \mathsf{BinTree} \to 1 + \mathsf{BinTree}$ such that $\mathsf{strees}\,\mathsf{node} = \mathsf{error}$, $\mathsf{strees}\,\mathsf{makebt}\,\langle t_1, t_2\rangle = \mathsf{inr}\langle t_1, t_2\rangle$
  $\mathsf{strees} = \lambda x.\mathsf{Rec}(\mathsf{mapbtree}, \lambda y.\mathsf{case}(y, u.\,\mathsf{inl}\,u, v.\,\mathsf{inr}\langle\mathsf{fst}(\mathsf{fst}\,v), \mathsf{fst}(\mathsf{snd}\,v)\rangle), x)$.
- Some functions on $\mathsf{BinTree}$
  - $\mathsf{isnode} : \mathsf{BinTree} \to \mathsf{Bool}$ decides if a given tree is a node.
    $\mathsf{isnode} = \lambda t.\mathsf{case}(\mathsf{in}^{-1}(\mathsf{mapbtree}, t), x.\mathsf{true}, y.\mathsf{false})$.
  - $\mathsf{nn} : \mathsf{BinTree} \to \mathsf{Nat}$ returns the number of nodes given by
    $\mathsf{nn}\,\mathsf{node} = 1,\ \mathsf{nn}(\mathsf{makebt}\langle t_1, t_2\rangle) = \mathsf{suc}(\mathsf{sum}\,(\mathsf{nn}\,t_1)\,(\mathsf{nn}\,t_2))$.
    $\mathsf{nn} = \lambda x.\mathsf{It}(\mathsf{mapbtree}, \lambda y.\mathsf{case}(y, u.1, v.\,\mathsf{suc}(\mathsf{sum}(\mathsf{fst}\,v)(\mathsf{snd}\,v)), x)$.

**Example 3.6** (unlabelled $A$-branching well-founded trees with successor)**.** The type $\mathsf{WFTreeS}\,A = \mu X.1+X+(A \to X)$ with constructor $\mathsf{in} : 1+\mathsf{WFTreeS}\,A+(A \to \mathsf{WFTreeS}\,A) \to \mathsf{WFTreeS}\,A$ encodes the following constructors:

- $\mathsf{node} : \mathsf{WFTreeS}\,A, \quad \mathsf{node} = \mathsf{in}(\mathsf{inl}\,\star)$.
- $\mathsf{succ} : \mathsf{WFTreeS}\,A \to \mathsf{WFTreeS}\,A \quad \mathsf{succ} = \lambda x.\,\mathsf{in}(\mathsf{inr}(\mathsf{inl}\,x))$.
- $\mathsf{makewft} : (A \to \mathsf{WFTreeS}\,A) \to \mathsf{WFTreeS}\,A \quad \mathsf{makewft} = \lambda f.\,\mathsf{in}(\mathsf{inr}(\mathsf{inr}\,f))$.

We discuss now the particular instance $\mathcal{O} = \mathsf{WFTreeS}\,\mathsf{Nat}$ which rises the type of Brouwer ordinals or Kleene's $\mathcal{O}$.

- Monotonicity witness:

$$\mathsf{mapord} = \lambda f.\lambda x.\mathsf{case}(x, y.\,\mathsf{inl}\,y, z.\mathsf{case}(z, u.\,\mathsf{inr}(\mathsf{inl}\,(f u)), v.\,\mathsf{inr}(\mathsf{inr}\,(\lambda n.f(v\,n))))).$$

- Constructors: in this particular case the above constructors are called $\mathsf{0}, \mathsf{succ}$ and $\mathsf{lim}$ respectively, where $\mathsf{lim}\,f$ refers to the least upper bound of the ordinals on the range of $f$.
- Destructor: a predecessor function $\mathsf{pred} : \mathcal{O} \to 1+\mathcal{O}+(\mathsf{Nat} \to \mathcal{O})$ such that $\mathsf{pred}\ 0 = \mathsf{error},\ \mathsf{pred}\,(\mathsf{succ}\,\alpha) = \mathsf{inr}(\mathsf{inl}\,\alpha)$ and $\mathsf{pred}\,(\mathsf{lim}\,f) = \mathsf{inr}(\mathsf{inr}(\lambda n.f\,n))$ can easily be defined by inductive inversion. However, if the predecessor of

a limit $\mathsf{lim}\, f$ is understood as a function mapping every natural number $n$ to the predecessor of its image under $f$, then it is not possible to give a definition neither by inversion nor by iteration. This task is easily achieved by primitive recursion on $\mathcal{O}$ as follows: $\mathsf{pred} = \lambda\alpha.\mathsf{Rec}(\mathsf{mapord}, s, \alpha)$ where $s = \lambda x.\mathsf{case}(x, y.\,\mathsf{inl}\, y, z.\mathsf{case}(z, u.\,\mathsf{inr}(\mathsf{inl}\,(\mathsf{fst}\, u)), v.\,\mathsf{inr}(\mathsf{inr}\,(\lambda n.\,\mathsf{in}(\mathsf{snd}(v\, n))))))$. With this definition the predecessor behaves as before for 0 and $\mathsf{suc}\,\alpha$ whereas $\mathsf{pred}(\mathsf{lim}\, f) = \lambda n.\,\mathsf{inr}(\mathsf{inr}(\mathsf{in}(\mathsf{pred}\,(f\, n))))$.

- Some functions on $\mathcal{O}$:
  - Ordinal sum: $\mathsf{sum} : \mathcal{O} \to \mathcal{O} \to \mathcal{O}$, $\mathsf{sum} = \lambda\alpha.\lambda x.\mathsf{It}(\mathsf{map}, s, x)$ where $s = [\lambda\_.\alpha, [\mathsf{succ}, \mathsf{lim}]]$. With this definition we have $\mathsf{sum}\,\alpha\, 0 = \alpha$, $\mathsf{sum}\,\alpha\,(\mathsf{succ}\,\beta) = \mathsf{succ}\,(\mathsf{sum}\,\alpha\,\beta)$ and $\mathsf{sum}\,\alpha\,(\mathsf{lim}\,\gamma) = \mathsf{lim}(\lambda n.\mathsf{sum}\,\alpha\,(\gamma\, n))$.
  - Ordinal product: $\mathsf{prod} : \mathcal{O} \to \mathcal{O} \to \mathcal{O}$, $\mathsf{prod} = \lambda\alpha.\lambda x.\mathsf{It}(\mathsf{map}, s, x)$ where $s = [\lambda\_.0, [\lambda\beta.\mathsf{sum}\,\beta\,\alpha, \mathsf{lim}]]$. With this definition we have $\mathsf{prod}\,\alpha\, 0 = 0$, $\mathsf{prod}\,\alpha\,(\mathsf{succ}\,\beta) = \mathsf{sum}\,(\mathsf{prod}\,\alpha\,\beta)\,\alpha$ and $\mathsf{prod}\,\alpha\,(\mathsf{lim}\,\gamma) = \mathsf{lim}\,(\lambda n.\mathsf{prod}\,\alpha\,(\gamma\, n))$.

Let us discuss now some examples of coinductive types representing codatatypes defined by their destructors.

**Example 3.7** (streams over $A$)**.** The type of streams or strictly infinite lists over a type $A$ is defined as $\mathsf{Stream}\, A = \nu X.A \times X$ with $\mathsf{out} : \mathsf{Stream}\, A \to A \times \mathsf{Stream}\, A$

- Canonical monotonicity witness: $\mathsf{mapstream} = \lambda f \lambda x.\langle \mathsf{fst}\, x, f(\mathsf{snd}\, x)\rangle$.
- Destructors:
  - $\mathsf{head} : \mathsf{Stream}\, A \to A$, $\mathsf{head} = \lambda x.\,\mathsf{fst}(\mathsf{out}\, x)$
  - $\mathsf{tail} : \mathsf{Stream}\, A \to \mathsf{Stream}\, A$, $\mathsf{tail} = \lambda x.\,\mathsf{snd}(\mathsf{out}\, x)$.
- Constructor $\mathsf{cons} : A \times \mathsf{Stream}\, A \to \mathsf{Stream}\, A$, given by $\mathsf{cons} = \lambda x.\,\mathsf{out}^{-1}(\mathsf{mapstream}, x)$.
- Some functions:
  - $\mathsf{cnt} : A \to \mathsf{Stream}\, A$ giving a stream of constants
    $\mathsf{head}(\mathsf{cnt}\, a) = a$, $\mathsf{tail}(\mathsf{cnt}\, a) = \mathsf{cnt}\, a$
    $\mathsf{cnt} = \lambda z.\mathsf{CoIt}(\mathsf{mapstream}, \lambda y.\langle y, y\rangle, z)$
  - $\mathsf{from} : \mathsf{Nat} \to \mathsf{Stream}\,\mathsf{Nat}$ returning the stream of naturals from the given one
    $\mathsf{head}(\mathsf{from}\, n) = n$, $\mathsf{tail}(\mathsf{from}\, n) = \mathsf{from}(\mathsf{suc}\, n)$
    $\mathsf{from} = \lambda z.\mathsf{CoIt}(\mathsf{mapstream}, \lambda y.\langle y, \mathsf{suc}\, y\rangle, z)$.
  - A map function $\mathsf{mapstr} : (A \to B) \times \mathsf{Stream}\, A \to \mathsf{Stream}\, B$ returning the stream resulting of applying $f$ to the elements in a given stream.
    $\mathsf{head}(\mathsf{mapstr}\langle f, s\rangle) = f\,(\mathsf{head}\, s)$, $\mathsf{tail}(\mathsf{mapstr}\langle f, s\rangle) = \mathsf{mapstr}\langle f, \mathsf{tail}\, s\rangle$.
    $\mathsf{mapstr} = \lambda z.\mathsf{CoIt}(\mathsf{mapstream}, \lambda p.\langle(\mathsf{fst}\, p)(\mathsf{head}(\mathsf{snd}\, p)), \langle \mathsf{fst}\, p, \mathsf{tail}(\mathsf{snd}\, p)\rangle\rangle, z)$. Observe that this function can be polymorphic with type $\forall X \forall Y.(X \to Y) \times \mathsf{Stream}\, X \to \mathsf{Stream}\, Y$.

Of course it would be more elegant to give a curried version of mapstr. From now on, whenever possible, we will give only curried functions.

The next example shows the limitations of coiteration and the expressivity of corecursion.

**Example 3.8** (conatural numbers (the ordinal $\omega+1$))**.** $\mathsf{CoNat} = \nu X.1 + X$

- Canonical monotonicity witness:
  $\mathsf{mapconat} = \lambda f \lambda x.\mathsf{case}(x, u.\,\mathsf{inl}\star, v.\,\mathsf{inr}(f\,v))$.
- Destructor: $\mathsf{pred} : \mathsf{CoNat} \to 1 + \mathsf{CoNat}$, $\mathsf{pred} = \mathsf{out}$. Observe that this is the predecessor function with error such that $\mathsf{pred}\ 0 \to^\star \mathsf{error}$.
- Constructors:
  - Zero: $0 : \mathsf{CoNat},\ 0 = \mathsf{out}^{-\mathbf{1}}(\mathsf{mapconat}, \mathsf{inl}\star)$.
  - Successor: $\mathsf{suc} : \mathsf{CoNat} \to \mathsf{CoNat},\ \mathsf{suc} = \lambda n.\,\mathsf{out}^{-\mathbf{1}}(\mathsf{mapconat}, \mathsf{inr}\,n)$.
  - Omega: to define the ordinal $\omega$, we first define a global element of $\mathsf{CoNat}$, $\omega^\dagger : 1 \to \mathsf{CoNat}$ by coiteration as: $\omega^\dagger = \lambda x.\mathsf{CoIt}(\mathsf{mapconat}, \mathsf{inr}, x)$. Omega is then defined as $\omega = \omega^\dagger\star$. With this definition we have $\mathsf{pred}(\mathsf{suc}\,\omega) \to^\star \mathsf{inr}\,\omega$ and also $\mathsf{pred}(\omega) \to^\star \mathsf{inr}\,\omega$.
- Some functions on $\mathsf{CoNat}$:
  - Test for zero: the function $\mathsf{iszero} :: \mathsf{CoNat} \to \mathsf{Bool}$ is defined as $\mathsf{iszero} = [\lambda\_.\mathsf{false}, \lambda\_.\mathsf{true}] \circ \mathsf{pred}$.
  - Coiterative sum of conaturals: the function $\oplus : \mathsf{CoNat} \to \mathsf{CoNat} \to \mathsf{CoNat}$ is defined as $n \oplus m = \mathsf{sum}\langle \mathsf{inr}\,n, \mathsf{inr}\,m\rangle$ where $\mathsf{sum} : (1+\mathsf{CoNat}) \times (1 + \mathsf{CoNat}) \to \mathsf{CoNat}$ is such that

$$\begin{array}{rcl} \mathsf{pred}(\mathsf{sum}\langle \mathsf{inr}\,0, \mathsf{inr}\,0\rangle) & = & \mathsf{error} \\ \mathsf{pred}(\mathsf{sum}\langle \mathsf{inr}\,0, \mathsf{inr}\,m\rangle) & = & \mathsf{inr}(\mathsf{sum}\langle \mathsf{inr}\,0, \mathsf{pred}\,m\rangle) \\ \mathsf{pred}(\mathsf{sum}\langle \mathsf{inr}\,n, \mathsf{inr}\,m\rangle) & = & \mathsf{inr}(\mathsf{sum}\langle \mathsf{pred}\,n, \mathsf{inr}\,m\rangle). \end{array}$$

    This function is defined as $\mathsf{sum} = \lambda z.\mathsf{CoIt}(\mathsf{mapconat}, s, z)$ where

$$\begin{array}{rcl} s\ x & = & \mathsf{if}\ \mathsf{iszero}(\mathsf{fst}\,x)\ \mathsf{then} \\ & & \quad \mathsf{if}\ \mathsf{iszero}(\mathsf{snd}\,x)\ \mathsf{then} \\ & & \quad\quad \mathsf{error} \\ & & \quad \mathsf{else}\ \ \mathsf{inr}\langle \mathsf{fst}\,x, \mathsf{pred}(\mathsf{snd}\,x)\rangle \\ & & \mathsf{else} \\ & & \quad \mathsf{inr}\langle \mathsf{pred}(\mathsf{fst}\,x), \mathsf{snd}\,x\rangle. \end{array}$$

    However, the operational semantics of this sum function is not satisfactory. For instance we only get $\mathsf{pred}(0 \oplus (\mathsf{suc}\,m)) \to^\star \mathsf{inr}(0 \oplus m)$ or $\mathsf{pred}(n \oplus \omega) \to^\star \mathsf{inr}(0 \oplus \omega)$ but neither $0 \oplus m \to^\star m$ nor $\mathsf{pred}(\omega \oplus m) \to^\star \mathsf{inr}\,\omega$. This could be arranged by directly defining functions $0 \oplus \_$ and $\omega \oplus \_$ as the identity $\lambda x x$ and the constant function $\lambda\_.\omega$, respectively. However there is no hope for defining a function $(\mathsf{suc}\,n) \oplus \_$ in a similar way, for its destruction by $\mathsf{pred}$ does not involve itself as required by the coiteration principle, but rather calls another function, namely $n \oplus \_$. Fortunately this function can be defined with corecursion.
  - Corecursive sum of conaturals: this time the sum is defined as $n \oplus m = \mathsf{sum}\,n\,m$ where a function $\mathsf{sum}\ \ n : \mathsf{CoNat} \to \mathsf{CoNat}$ is defined for each conatural number $n$ as follows. The cases for $0$ and $\omega$ are defined directly as $\mathsf{sum}\,0 = \mathsf{Id}$ and $\mathsf{sum}\,\omega = \lambda\_.\omega$ and for a successor we use the

corecursion principle, defining $\mathsf{sum}(\mathsf{suc}\, n) = \lambda z.\mathsf{CoRec}(\mathsf{mapconat}, s, z)$ where $s : \mathsf{CoNat} \to 1 + (\mathsf{CoNat} + \mathsf{CoNat})$ is given by

$$\begin{array}{rcl} s\ x & = & \mathsf{if}\ \mathsf{iszero}\, x\ \mathsf{then} \\ & & \quad \mathsf{inr}(\mathsf{inl}\, n) \\ & & \mathsf{else} \\ & & \quad \mathsf{inr}(\mathsf{inl}(\mathsf{sum}\, n\, x)). \end{array}$$

It is easily verified that now the following specification holds:

$$\mathsf{pred}(0 \oplus m) = \mathsf{pred}\, m \qquad \mathsf{pred}((\mathsf{suc}\, n) \oplus m) = \mathsf{inr}(n \oplus m) \qquad \mathsf{pred}(\omega \oplus m) = \mathsf{inr}\, \omega.$$

The next example is related to example 2.4 but in this case we can have the empty list.

**Example 3.9** (coinductive lists over $A$)**.** The type $\mathsf{CoList}\, A = \nu X.1 + A \times X$ entails potentially infinite lists over $A$.

- Canonical monotonicity witness: $\mathsf{mapcolist} = \mathsf{maplist}$, see Example 3.4.
- Destructors:
  - $\mathsf{head} : \mathsf{CoList}\, A \to 1 + \mathsf{CoList}\, A$, $\mathsf{head} = \lambda x.\mathsf{case}(\mathsf{out}\, x, y.\, \mathsf{inl}\, y, z.\, \mathsf{inr}(\mathsf{fst}\, z))$.
  - $\mathsf{tail} : \mathsf{CoList}\, A \to 1 + \mathsf{CoList}\, A$, $\mathsf{tail} = \lambda x.\mathsf{case}(\mathsf{out}\, x, y.\, \mathsf{inl}\, y, z.\, \mathsf{inr}(\mathsf{snd}\, z))$.
- Constructors: $\mathsf{nil} : \mathsf{CoList}\, A$ is defined as $\mathsf{nil} = \mathsf{out}^{-1}(\mathsf{mapcolist}, \mathsf{inl}\, \star)$ whereas $\mathsf{cons} : A \times \mathsf{CoList}\, A \to \mathsf{CoList}\, A$ is given by $\mathsf{cons} = \lambda x.\, \mathsf{out}^{-1}(\mathsf{mapcolist}, \mathsf{inr}\, x)$.
- Some functions:
  - Test for nil: $\mathsf{isnil} : \mathsf{CoList}\, A \to \mathsf{Bool}$ is defined by $\mathsf{isnil} = \lambda x.\mathsf{case}(\mathsf{out}\, x, y.\mathsf{true}, z.\mathsf{false})$.
  - Coiterative append: $\mathsf{coapp} : \mathsf{CoList}\, A \times \mathsf{CoList}\, A \to \mathsf{CoList}\, A$, $\mathsf{coapp} = \lambda z.\mathsf{CoIt}(\mathsf{mapcolist}, s, z)$ where

$$\begin{array}{rcl} s\ x & = & \mathsf{if}\, \mathsf{isnil}(\mathsf{fst}\, x)\ \mathsf{then} \\ & & \quad \mathsf{if}\, \mathsf{isnil}(\mathsf{snd}\, x)\ \mathsf{then} \\ & & \quad\quad \mathsf{error} \\ & & \quad \mathsf{else}\ \mathsf{inr}\langle \mathsf{head}(\mathsf{snd}\, x), \langle \mathsf{nil}, \mathsf{tail}(\mathsf{snd}\, x)\rangle\rangle \\ & & \mathsf{else} \\ & & \quad \mathsf{inr}\langle \mathsf{head}(\mathsf{fst}\, x), \langle \mathsf{tail}(\mathsf{fst}\, x), \mathsf{snd}\, x\rangle\rangle. \end{array}$$

    However with this definition we get a very inefficient behavior corresponding to the definition $\mathsf{coapp}\langle \mathsf{nil}, \mathsf{cons}\langle x, xs\rangle\rangle = \mathsf{cons}\langle x, \mathsf{coapp}\langle \mathsf{nil}, xs\rangle\rangle$, which forces coapp to continue executing until a copy of $xs$ is constructed. We cannot do better with coiteration, for this principle oblige us to call coapp with a new seed, namely $\langle \mathsf{nil}, xs\rangle$, to keep coiterating. Instead, we would like to return immediately the list $xs$. This can be done with primitive corecursion.
  - Corecursive append: this time we define, for every given colist $xs$, a function $\mathsf{coapp}\, xs : \mathsf{CoList}\, A \to \mathsf{CoList}\, A$. If $xs$ is nil then we can use

corecursion to get $\mathsf{coapp}\,\mathsf{nil}\,ys = ys$ or simply define $\mathsf{coapp}\,\mathsf{nil} = \mathsf{Id}$. For the case $\mathsf{coapp}\,xs$ where $xs$ is not empty, we define $\mathsf{coapp}\,xs = \lambda z.\mathsf{CoRec}(\mathsf{mapcolist}, s, z)$ by taking the step function $s : \mathsf{CoList}\,A \to 1 + A \times (\mathsf{CoList}\,A + \mathsf{CoList}\,A)$ defined as follows:

$$\begin{array}{lcl} s\,x & = & \mathsf{if}\ \ \mathsf{isnil}\,x\ \mathsf{then} \\ & & \quad \mathsf{error} \\ & & \mathsf{else} \\ & & \quad \mathsf{inr}\langle \mathsf{head}\,xs, \mathsf{inl}(\mathsf{coapp}(\mathsf{tail}\,xs)\,x)\rangle. \end{array}$$

From all the above examples we can observe the heavy use of injections and projections which complicate both the definitions of map witnesses and of functions in general, for the constructors or destructors of a type are encoded and not directly available. After the proof of strong normalization of this system, we will present an improved system which allows the constructors and destructors to be expressed directly and therefore modularizes the definition of types and functions.

### 3.3. A word on positivity

The reader can confirm that the above examples are in fact positive, that is, the variable $X$ occurs only on positive positions in the type $F$. An immediate question arises: what are then the advantages of using full monotonicity?

Above all, positive systems are incompatible with Curry-style in the following sense: in a positive system a functor is a pair $\langle \lambda X.F, \mathsf{map}_{\lambda X.F}\rangle$ where $X$ occurs only in positive positions in $F$ and the monotonicity witness is fixed and defined according to the shape of $F$, essentially by the rules given in Appendix B. Observe that this term is annotated by the type expression $\lambda X.F$. The (co)inductive types are only allowed if the positivity restriction holds, therefore there is no necessity to attach the witness in the typing rules. For example the $(\mu E)$ rule becomes

$$\frac{\Gamma \vdash t : \mu X.F \qquad \Gamma \vdash s : F[X := B] \to B}{\Gamma \vdash \mathsf{It}(s,t) : B}$$

and the corresponding reduction rule becomes:

$$\mathsf{It}(s, \mathsf{in}\,t) \mapsto_\beta s\Big(\mathsf{map}\big(\lambda x.\mathsf{It}(s,x)\big)t\Big).$$

But there is no way to recover the specific term map only from the terms. It is mandatory either to look for the adequate instance of $(\mu E)$ or to annotate some term to recover the map term. In the first case we would get a conditional term rewrite system very hard to handle from the metatheoretical point of view. In the second case we would obviously get a Church-style system, and the rule becomes, for instance

$$\mathsf{It}_{\mu X.F}(s, \mathsf{in}\,t) \mapsto_\beta s\Big(\mathsf{map}_{\lambda X.F}\big(\lambda x.\mathsf{It}(s,x)\big)t\Big).$$

Other answers in favor of monotonicity are:

- Specific monotonicity witnesses are not involved in proofs, we can even have hypothetical monotonicity, *i.e.* just an additional assumption $x : F\,\mathsf{mon}\,X$ in our context. Therefore the generality of our approach simplifies proofs.
- For higher-order systems there is no fixed concept of positivity. With full monotonicity we can generalize directly the systems presented in this work, this has been done in [1,22]. Moreover, sometimes different witnesses are useful for programming, see the example on power list reverse in [1].

Next, we prove a strong version of safety for MICT by proving termination and type-preservation.

## 4. Strong normalization for MICT

In this section we show the termination property for MICT, proving that every typable term in MICT strongly normalizes by means of a variation of the well-known Tait's method [34], using the so-called saturated sets which are a variant of Girard's *candidats de reducibilité* [10,11]. This method characterizes the typable strongly normalizing terms in a syntax-directed way and modularizes in a convenient way the normalization proof. The proof presented here, specifically the constructions on saturated sets, is based on proofs given in [19] for related inductive type systems in Church-style. The coinductive constructions are, to the best of our knowledge, new.

**Definition 4.1.** A term $t$ is strongly normalizing with respect to a reduction relation $\to$ if there is no infinite reduction sequence $t \to t_1 \to t_2 \to \dots$

Equivalently, the set $\mathsf{sn}$ of strongly normalizing terms can be defined inductively as follows:

$$\frac{\text{For all } t', t \to t' \text{ implies } t' \in \mathsf{sn}}{t \in \mathsf{sn}}.$$

Thus, the set $\mathsf{sn}$ corresponds to the accessible or well-founded part of the relation $\to$.

The syntactical concept of evaluation context, defined next, is the main tool to characterize the set of strongly normalizing terms without recurring to the reduction relation.

**Definition 4.2.** Let $\bullet$ be a new symbol. An elimination is an expression of one of the following forms:

$$\bullet s,\ \mathsf{case}(\bullet, x.t, y.r),\ \mathsf{fst}\,\bullet,\ \mathsf{snd}\,\bullet,\ \mathsf{It}(m, s, \bullet),\ \mathsf{Rec}(m, s, \bullet),\ \mathsf{in}^{-1}(m, \bullet),\ \mathsf{out}\,\bullet$$

eliminations are denoted always with the letter $e$.

The evaluation contexts, called multiple eliminations in [23], are generated by eliminations. An evaluation context is like a term with a hole somewhere inside it, into which we can plug a term to obtain a new term.

**Definition 4.3.** An evaluation context is defined as follows:

$$E ::= \bullet \mid e[\bullet := E]$$

where the substitution $e[\bullet := E]$ is defined as if $\bullet$ were a term variable; in fact, textual substitution suffices, for $\bullet$ is never below a binder. From now on we will use $E[r]$ to denote $E[\bullet := r]$.

Next, we define the set of terms $\mathsf{SN}$ which characterize strong normalizability via a syntax-directed inductive definition.

**Definition 4.4.** The set $\mathsf{SN}$ is inductively defined as follows:

$$\frac{}{x \in \mathsf{SN}} \qquad \frac{E[x], s \in \mathsf{SN}}{E[x]s \in \mathsf{SN}} \qquad \frac{E[x], s, t \in \mathsf{SN}}{\mathsf{case}(E[x], x.s, y.t) \in \mathsf{SN}}$$

$$\frac{E[x] \in \mathsf{SN}}{\mathsf{fst}(E[x]) \in \mathsf{SN}} \qquad \frac{E[x] \in \mathsf{SN}}{\mathsf{snd}(E[x]) \in \mathsf{SN}}$$

$$\frac{r \in \mathsf{SN}}{\lambda x.r \in \mathsf{SN}} \qquad \frac{E\big[r[x := s]\big], s \in \mathsf{SN}}{E[(\lambda x.r)s] \in \mathsf{SN}} \qquad \frac{t \in \mathsf{SN}}{\mathsf{inl}\, t \in \mathsf{SN}} \qquad \frac{t \in \mathsf{SN}}{\mathsf{inr}\, t \in \mathsf{SN}}$$

$$\frac{E\big[r[x := t]\big], s, t \in \mathsf{SN}}{E[\mathsf{case}(\mathsf{inl}\, t, x.r, y.s)] \in \mathsf{SN}} \qquad \frac{E\big[s[y := t]\big], r, t \in \mathsf{SN}}{E\big[\mathsf{case}(\mathsf{inr}\, t, x.r, y.s)\big] \in \mathsf{SN}}$$

$$\frac{r, s \in \mathsf{SN}}{\langle r, s\rangle \in \mathsf{SN}} \qquad \frac{E[r], s \in \mathsf{SN}}{E[\mathsf{fst}\langle r, s\rangle] \in \mathsf{SN}} \qquad \frac{E[s], r \in \mathsf{SN}}{E[\mathsf{snd}\langle r, s\rangle] \in \mathsf{SN}}$$

$$\frac{m, s, E\big[x\big] \in \mathsf{SN}}{\mathsf{It}(m, s, E\big[x\big]) \in \mathsf{SN}} \qquad \frac{m, s, E\big[x\big] \in \mathsf{SN}}{\mathsf{Rec}(m, s, E\big[x\big]) \in \mathsf{SN}} \qquad \frac{m, E\big[x\big] \in \mathsf{SN}}{\mathsf{in}^{-1}(m, E\big[x\big]) \in \mathsf{SN}}$$

$$\frac{E\big[x\big] \in \mathsf{SN}}{\mathsf{out}\, E\big[x\big] \in \mathsf{SN}}$$

$$\frac{t \in \mathsf{SN}}{\mathsf{in}\, t \in \mathsf{SN}} \qquad \frac{E\big[s\Big(m\big(\lambda x.\mathsf{It}(m, s, x)\big)t\Big)\big] \in \mathsf{SN}}{E\big[\mathsf{It}(m, s, \mathsf{in}\, t)\big] \in \mathsf{SN}}$$

$$\frac{E\big[s\Big(m\big(\langle \mathsf{Id}, \lambda x.\mathsf{Rec}(m, s, x)\rangle\big)t\Big)\big] \in \mathsf{SN}}{E\big[\mathsf{Rec}(m, s, \mathsf{in}\, t)\big] \in \mathsf{SN}} \qquad \frac{E\big[m(\lambda z z)t\big] \in \mathsf{SN}}{E\big[\,\mathsf{in}^{-1}(m, \mathsf{in}\, t)\big] \in \mathsf{SN}}$$

$$\frac{m, s, t \in \mathsf{SN}}{\mathsf{CoIt}(m, s, t) \in \mathsf{SN}} \qquad \frac{m, s, t \in \mathsf{SN}}{\mathsf{CoRec}(m, s, t) \in \mathsf{SN}} \qquad \frac{m, t \in \mathsf{SN}}{\mathsf{out}^{-1}(m, t) \in \mathsf{SN}}$$

$$\frac{E\big[m\big(\lambda z.\mathsf{CoIt}(m, s, z)\big)(st)\big] \in \mathsf{SN}}{E\big[\,\mathsf{out}\,\mathsf{CoIt}(m, s, t)\big] \in \mathsf{SN}} \qquad \frac{E\big[m\big([\mathsf{Id}, \lambda z.\mathsf{CoRec}(m, s, z)]\big)(st)\big] \in \mathsf{SN}}{E\big[\,\mathsf{out}\,\mathsf{CoRec}(m, s, t)\big] \in \mathsf{SN}}$$

$$\frac{E\big[m(\lambda zz)t\big] \in \mathsf{SN}}{E\big[\,\mathsf{out}\,\mathsf{out}^{-\mathbf{1}}(m,t)\big] \in \mathsf{SN}}.$$

This definition of $\mathsf{SN}$ captures exhaustively the closure of terms under reduction without referring to the reduction relation itself. Moreover, it is based only on the typing rules; each expression (term or elimination) on the conclusion of a rule can only be derived with one of the typing rules.

**Proposition 4.1.** *The defining rules of* $\mathsf{SN}$ *are sound with respect to the reduction relation. That is,* $\mathsf{SN} \subseteq \mathsf{sn}$.

*Proof.* Several routinary inductions show that $\mathsf{sn}$ is closed under all the defining rules of $\mathsf{SN}$. Therefore the claim follows by minimality of $\mathsf{SN}$. □

### 4.1. Saturated sets

The proof now follows as usual by defining the subsets of terms which will serve as reducibility candidates. These so-called saturated sets have good closure properties and are modeled after $\mathsf{SN}$.

**Definition 4.5** (saturated set)**.** A set of terms $\mathcal{M}$ is saturated if and only if $\mathcal{M} \subseteq \mathsf{SN}$ and the following closure conditions hold:

$$\frac{E[x] \in \mathsf{SN}}{E[x] \in \mathcal{M}}$$

$$\frac{E\big[r[x := s]\big] \in \mathcal{M} \quad s \in \mathsf{SN}}{E[(\lambda x.r)s] \in \mathcal{M}}$$

$$\frac{E\big[r[x := t]\big], t \in \mathcal{M} \quad s \in \mathsf{SN}}{E[\mathsf{case}(\mathsf{inl}\,t, x.r, y.s)] \in \mathcal{M}} \qquad \frac{E\big[s[y := t]\big], t \in \mathcal{M} \quad r \in \mathsf{SN}}{E\big[\mathsf{case}(\mathsf{inr}\,t, x.r, y.s)\big] \in \mathcal{M}}$$

$$\frac{E[r] \in \mathcal{M} \quad s \in \mathsf{SN}}{E[\mathsf{fst}\langle r, s\rangle] \in \mathcal{M}} \qquad \frac{E[s] \in \mathcal{M} \quad r \in \mathsf{SN}}{E[\mathsf{snd}\langle r, s\rangle] \in \mathcal{M}}$$

$$\frac{E\big[s\Big(m\big(\lambda x.\mathsf{It}(m, s, x)\big)t\Big)\big] \in \mathcal{M}}{E\big[\mathsf{It}(m, s, \mathsf{in}\,t)\big] \in \mathcal{M}}$$

$$\frac{E\big[s\Big(m\big(\langle \mathsf{Id}, \lambda x.\mathsf{Rec}(m, s, x)\rangle\big)t\Big)\big] \in \mathcal{M}}{E\big[\mathsf{Rec}(m, s, \mathsf{in}\,t)\big] \in \mathcal{M}} \qquad \frac{E\big[m(\lambda zz)t\big] \in \mathcal{M}}{E\big[\,\mathsf{in}^{-\mathbf{1}}(m, \mathsf{in}\,t)\big] \in \mathcal{M}}$$

$$\frac{E\big[m\big(\lambda z.\mathsf{CoIt}(m, s, z)\big)(st)\big] \in \mathcal{M}}{E\big[\,\mathsf{out}\,\mathsf{CoIt}(m, s, t)\big] \in \mathcal{M}} \qquad \frac{E\big[m\big([\mathsf{Id}, \lambda z.\mathsf{CoRec}(m, s, z)]\big)(st)\big] \in \mathcal{M}}{E\big[\,\mathsf{out}\,\mathsf{CoRec}(m, s, t)\big] \in \mathcal{M}}$$

$$\frac{E\big[m(\lambda zz)t\big] \in \mathcal{M}}{E\big[\,\mathsf{out}\,\mathsf{out}^{-\mathbf{1}}(m, t)\big] \in \mathcal{M}}$$

the set of saturated sets will be denoted with $\mathsf{SAT}$.

The saturated sets are needed to recursively define the so-called predicates of strong computability starting from a candidate assignment, which is an assignment of saturated sets for type variables.

**Proposition 4.2.** $\mathsf{SN} \in \mathsf{SAT}$ *and* $\mathsf{SAT}$ *is closed under intersection.*

*Proof.* Straightforward □

**Definition 4.6.** Given a set of terms $M$ we define the saturated closure of $M$ as follows:

$$\mathsf{cl}(M) = \bigcap \{\mathcal{N} \in \mathsf{SAT} \mid M \cap \mathsf{SN} \subseteq \mathcal{N}\}$$

$\mathsf{cl}(M)$ is the least saturated superset of $M \cap \mathsf{SN}$. Observe that $M \subseteq \mathsf{cl}(M)$ if and only if $M \subseteq \mathsf{SN}$.

## 4.2. Constructions on saturated sets

The constructions for saturated sets corresponding to the type constructors of the system are central to the proof of strong normalization. In order for the proof to work, these constructions must be sound with respect to the typing rules of the system. In the next sections we define the constructions and prove their soundness. Let us start with the cases for function, sum and product types.

**Definition 4.7.** Given $\mathcal{M}, \mathcal{N} \in \mathsf{SAT}$, we define the following sets:

$$\begin{aligned}
\mathsf{S}_x(\mathcal{M},\mathcal{N}) &= \{t \mid \forall s \in \mathcal{M}.\, t[x := s] \in \mathcal{N}\} \\
\mathcal{I}_{\to}(\mathcal{M},\mathcal{N}) &= \{\lambda x.t \mid t \in \mathsf{S}_x(\mathcal{M},\mathcal{N})\} \\
\mathcal{I}_{+}(\mathcal{M},\mathcal{N}) &= \{\mathsf{inl}\, t \mid t \in \mathcal{M}\} \cup \{\mathsf{inr}\, t \mid t \in \mathcal{N}\} \\
\mathcal{I}_{\times}(\mathcal{M},\mathcal{N}) &= \{\langle s,t\rangle \mid s \in \mathcal{M} \text{ and } t \in \mathcal{N}\} \\
\mathcal{M} + \mathcal{N} &= \mathsf{cl}(\mathcal{I}_{+}(\mathcal{M},\mathcal{N})) \\
\mathcal{M} \times \mathcal{N} &= \mathsf{cl}(\mathcal{I}_{\times}(\mathcal{M},\mathcal{N})) \\
\mathcal{M} \to \mathcal{N} &= \mathsf{cl}(\mathcal{I}_{\to}(\mathcal{M},\mathcal{N})).
\end{aligned}$$

The following lemmas will be needed later.

**Lemma 4.1.** *For all* $\mathcal{P},\mathcal{Q},\mathcal{N} \in \mathsf{SAT}$. *If* $\mathcal{P} \subseteq \mathcal{Q}$ *then* $\mathcal{Q} \to \mathcal{N} \subseteq \mathcal{P} \to \mathcal{N}$.

*Proof.* Assuming $\mathcal{P} \subseteq \mathcal{Q}$, it suffices to show that $\mathcal{I}_{\to}(\mathcal{Q},\mathcal{N}) \cap \mathsf{SN} = \mathcal{I}_{\to}(\mathcal{Q},\mathcal{N}) \subseteq \mathcal{P} \to \mathcal{N}$, since this implies the needed conclusion. Take $\lambda x.t \in \mathcal{I}_{\to}(\mathcal{Q},\mathcal{N})$, *i.e.*, $t \in \mathsf{S}_x(\mathcal{Q},\mathcal{N})$. To show $\lambda x.t \in \mathcal{P} \to \mathcal{N}$ it suffices to prove that $t \in \mathsf{S}_x(\mathcal{P},\mathcal{N})$. Therefore we take $p \in \mathcal{P}$ and show that $t[x := p] \in \mathcal{N}$, but this is clear from the fact that $t \in \mathsf{S}_x(\mathcal{Q},\mathcal{N})$, for by assumption we also have $p \in \mathcal{Q}$. □

**Lemma 4.2.** *For all* $\mathcal{P},\mathcal{Q},\mathcal{N} \in \mathsf{SAT}$. *If* $\mathcal{P} \subseteq \mathcal{Q}$ *then* $\mathcal{N} \to \mathcal{P} \subseteq \mathcal{N} \to \mathcal{Q}$.

*Proof.* Assuming $\mathcal{P} \subseteq \mathcal{Q}$, it suffices to show that $\mathcal{I}_{\to}(\mathcal{N},\mathcal{P}) \cap \mathsf{SN} = \mathcal{I}_{\to}(\mathcal{N},\mathcal{P}) \subseteq \mathcal{I}_{\to}(\mathcal{N},\mathcal{Q})$, since this implies the conclusion of the lemma. Take $\lambda x.t \in \mathcal{I}_{\to}(\mathcal{N},\mathcal{P})$, *i.e.*, $t \in \mathsf{S}_x(\mathcal{N},\mathcal{P})$. Therefore we have $\forall s \in \mathcal{N}.t[x := s] \in \mathcal{P}$ which by assumption implies $\forall s \in \mathcal{N}.t[x := s] \in \mathcal{Q}$. Therefore $t \in \mathsf{S}_x(\mathcal{N},\mathcal{Q})$ which yields $\lambda x.t \in \mathcal{I}_{\to}(\mathcal{N},\mathcal{Q})$. □

Next, we prove that the above constructions are sound with respect to the typing rules for sums, products and function types.

**Proposition 4.3** (soundness of the constructions)**.** *Assume* $\mathcal{M},\mathcal{N},\mathcal{P} \in \mathsf{SAT}$, *then*

(1) *If* $s \in \mathcal{M}$ *and* $t \in \mathcal{N}$ *then* $\langle s,t\rangle \in \mathcal{M} \times \mathcal{N}$.
(2) *If* $r \in \mathcal{M} \times \mathcal{N}$ *then* $\mathsf{fst}\, r \in \mathcal{M}$ *and* $\mathsf{snd}\, r \in \mathcal{N}$.
(3) *If* $t \in \mathcal{M}$ *then* $\mathsf{inl}\, t \in \mathcal{M} + \mathcal{N}$.
(4) *If* $t \in \mathcal{N}$ *then* $\mathsf{inr}\, t \in \mathcal{M} + \mathcal{N}$.
(5) *If* $r \in \mathcal{M} + \mathcal{N}$, $s \in \mathsf{S}_x(\mathcal{M},\mathcal{P})$, $t \in \mathsf{S}_y(\mathcal{N},\mathcal{P})$ *then* $\mathsf{case}(r, x.s, y.t) \in \mathcal{P}$.
(6) *If* $t \in \mathsf{S}_x(\mathcal{M},\mathcal{N})$ *then* $\lambda x.t \in \mathcal{M} \to \mathcal{N}$.
(7) *If* $r \in \mathcal{M} \to \mathcal{N}$ *and* $s \in \mathcal{M}$ *then* $rs \in \mathcal{N}$.

*Proof.* For part (1) clearly $\mathcal{I}_{\times}(\mathcal{M},\mathcal{N}) \subseteq \mathsf{SN}$ and therefore $\mathcal{I}_{\times}(\mathcal{M},\mathcal{N}) \subseteq \mathcal{M} \times \mathcal{N}$. The claim is now obvious. For part (2) set $\mathcal{E}_{\times}(\mathcal{M},\mathcal{N}) = \{r \in \mathsf{SN} \mid \mathsf{fst}\, r \in \mathcal{M} \text{ and } \mathsf{snd}\, r \in \mathcal{N}\} \subseteq \mathsf{SN}$, we will show that $\mathcal{M} \times \mathcal{N} \subseteq \mathcal{E}_{\times}(\mathcal{M},\mathcal{N})$ which will prove the claim. By definition of $\mathcal{M} \times \mathcal{N}$ it suffices to prove both, $\mathcal{E}_{\times}(\mathcal{M},\mathcal{N}) \in \mathsf{SAT}$, and $\mathcal{I}_{\times}(\mathcal{M},\mathcal{N}) \subseteq \mathcal{E}_{\times}(\mathcal{M},\mathcal{N})$. Let us start with the former inclusion; we have to show that $\mathcal{E}_{\times}(\mathcal{M},\mathcal{N})$ is closed under the rules for saturated sets. As an example take $E\big[r[x := s]\big] \in \mathcal{E}_{\times}(\mathcal{M},\mathcal{N})$ and $s \in \mathsf{SN}$. We have to show that $E[(\lambda x.r)s] \in \mathcal{E}_{\times}(\mathcal{M},\mathcal{N})$. $E\big[r[x := s]\big] \in \mathcal{E}_{\times}(\mathcal{M},\mathcal{N})$ implies $\mathsf{fst}(E\big[r[x := s]\big]) \in \mathcal{M}$ and $\mathsf{snd}(E\big[r[x := s]\big]) \in \mathcal{N}$. Observe that $\mathsf{fst}(E\big[r[x := s]\big]) = (\mathsf{fst}\,\bullet)\big[\bullet := E\big[r[x := s]\big]\big] = (\mathsf{fst}\,\bullet)[\bullet := E]\big[r[x := s]\big]$ and that $(\mathsf{fst}\,\bullet)[\bullet := E]$ is again an evaluation context, say $E'$, therefore we have $E'\big[r[x := s]\big] \in \mathcal{M}$ and, as $s \in \mathsf{SN}$ and $\mathcal{M} \in \mathsf{SAT}$, we get $E'[(\lambda x.r)s] \in \mathsf{SN}$, *i.e.*, $\mathsf{fst}(E[(\lambda x.r)s]) \in \mathcal{M}$. Analogously, we show that $\mathsf{snd}(E[(\lambda x.r)s]) \in \mathcal{N}$. Therefore $E[(\lambda x.r)s] \in \mathcal{E}_{\times}(\mathcal{M},\mathcal{N})$. The other rules for saturated sets are proved similarly.

To prove that $\mathcal{I}_{\times}(\mathcal{M},\mathcal{N}) \subseteq \mathcal{E}_{\times}(\mathcal{M},\mathcal{N})$ take $\langle s,t\rangle \in \mathcal{I}_{\times}(\mathcal{M},\mathcal{N})$, then $s \in \mathcal{M}$ and $t \in \mathcal{N}$. Observe that $s = \bullet[s] \in \mathcal{M}$ is an evaluation context and $t \in \mathsf{SN}$, for $\mathcal{N} \subseteq \mathsf{SN}$. Therefore, as $\mathcal{M} \in \mathsf{SAT}$, we have $\bullet[\mathsf{fst}\langle s,t\rangle] \in \mathcal{M}$, that is, $\mathsf{fst}\langle s,t\rangle \in \mathcal{M}$ and analogously $\mathsf{snd}\langle s,t\rangle \in \mathcal{N}$.

Parts (3), (4) and (6) are proved analogously to part (1) using instead $\mathcal{I}_+(\mathcal{M},\mathcal{N})$ and $\mathcal{I}_\to(\mathcal{M},\mathcal{N})$. To show parts (5) and (7) we use a similar argument to part (2) using $\mathcal{E}_+(\mathcal{M},\mathcal{N}) = \{r \in \mathsf{SN} \mid \forall \mathcal{P}\, \forall x\, \forall s \in \mathsf{S}_x(\mathcal{M},\mathcal{P})\, \forall y\, \forall t \in \mathsf{S}_y(\mathcal{N},\mathcal{P}).\, \mathsf{case}(r, x.s, y.t) \in \mathcal{P}\}$ and $\mathcal{E}_\to(\mathcal{M},\mathcal{N}) = \{r \in \mathsf{SN} \mid \forall s \in \mathcal{M}.\, rs \in \mathcal{N}\}$, respectively. □

We continue our proof by building saturated sets corresponding to terms related with (co)inductive types.

### 4.3. The Knaster-Tarski theorem

Our next task is to carry on with the constructions on saturated sets corresponding to (co)inductive types. To this purpose we will make use of the well-known Knaster-Tarski fixed-point theorem. For the sake of self-containment we recall the basics about it here.

**Definition 4.8.** Let $\langle \mathcal{L}, \sqsubseteq \rangle$ be a partially ordered set. If every set $M \subseteq \mathcal{L}$ has an infimum (greatest lower bound) in $\mathcal{L}$, denoted $\bigsqcap M$, we say that $\langle L, \sqsubseteq, \bigsqcap \rangle$ is a complete lattice.

In case $\mathcal{L}$ is a complete lattice, it is also truth that every $M \subseteq \mathcal{L}$ has a supremum (least upper bound), denoted $\bigsqcup M$ and defined as $\bigsqcup M = \bigsqcap\{x \in \mathcal{L} \mid \forall y \in M.y \sqsubseteq x\}$.

**Theorem 4.1** (Knaster-Tarski)**.** *Let $\langle \mathcal{L}, \sqsubseteq, \bigsqcap \rangle$ be a complete lattice and $\Phi : \mathcal{L} \to \mathcal{L}$ monotone (i.e., if $X \sqsubseteq Y$ then $\Phi(X) \sqsubseteq \Phi(Y)$). Then*

- *$\Phi$ has a least fixed point, denoted $\mu(\Phi)$ given by*

$$\mathsf{lfp}(\Phi) = \bigsqcap\{X \in \mathcal{L} \mid X \sqsubseteq \Phi(X)\};$$

- *$\Phi$ has a greatest fixed point, denoted $\nu(\Phi)$ given by*

$$\mathsf{gfp}(\Phi) = \bigsqcap\{X \in \mathcal{L} \mid \Phi(X) \sqsubseteq X\}.$$

*Proof.* See [5] □

The following corollary provides us with several methods of proof that will be used later.

**Corollary 4.1.** *Let $\Phi : \mathcal{L} \to \mathcal{L}$ be a monotone operator on a complete lattice $\mathcal{L}$. The following holds for every $M \in \mathcal{L}$.*

- *Induction: if $\Phi(M) \sqsubseteq M$ then $\mathsf{lfp}(\Phi) \sqsubseteq M$.*
- *Extended induction: if $\Phi\big(\mathsf{lfp}(\Phi) \bigsqcap M\big) \sqsubseteq M$ then $\mathsf{lfp}(\Phi) \sqsubseteq M$.*
- *Coinduction: if $M \sqsubseteq \Phi(M)$ then $M \sqsubseteq \mathsf{gfp}(\Phi)$.*
- *Extended coinduction: if $M \sqsubseteq \Phi\big(\mathsf{gfp}(\Phi) \bigsqcup M\big)$ then $M \sqsubseteq \mathsf{gfp}(\Phi)$.*

*Proof.* Straightforward. □

It is important to remark that the partially ordered set $\langle \mathsf{SAT}, \subseteq, \bigcap \rangle$ is a complete lattice due to Proposition 4.2 and therefore every monotone operator $\Phi : \mathsf{SAT} \to \mathsf{SAT}$ has a least and a greatest fixed-point.

4.3.1. *Saturated sets for inductive types*

The main goal of this section is to define saturated sets corresponding to inductive types. To this purpose we will make use of the Knaster-Tarski theorem to guarantee the existence of a least fixed point corresponding to an operator $\Phi : \mathsf{SAT} \to \mathsf{SAT}$. The constructions and methodology presented in this section are based on the ones given in [19] for related systems in Church-style. Related constructions for arbitrary fixed points can be found in [23].

From now on, we fix $\Phi : \mathsf{SAT} \to \mathsf{SAT}$.

**Definition 4.9.** Given $\mathcal{M} \in \mathsf{SAT}$ we define $\mathcal{I}_\mu(\mathcal{M}) = \{\mathsf{in}\, r \mid r \in \Phi(\mathcal{M})\}$ and $\Psi_I : \mathsf{SAT} \to \mathsf{SAT}$ as $\Psi_I(\mathcal{M}) = \mathsf{cl}(\mathcal{I}_\mu(\mathcal{M}))$.

Observe that we are not assuming that the operator $\Phi$ is monotone, therefore we cannot prove either that $\Psi_I$ is monotone. The reason not to restrict ourselves to a monotone operator $\Phi$ will be made clear later. This is an essential difference with the treatment in Definition 27, page 221 in [23].

Given $\Phi$ we define a saturated set denoted $\mu(\Phi)$, as the least fixed point of a monotone operator associated to $\Phi$ as follows: define $\mathsf{mon}(\Phi) = \bigcap_{\mathcal{P},\mathcal{Q} \in \mathsf{SAT}} (\mathcal{P} \to \mathcal{Q}) \to (\Phi(\mathcal{P}) \to \Phi(\mathcal{Q}))$, which is a saturated set due to Proposition 4.2, and $\Phi^{\supseteq} : \mathsf{SAT} \to \mathcal{P}(\mathsf{SN})$ as:

$$\Phi^{\supseteq}(\mathcal{M}) = \{t \in \mathsf{SN} \mid \forall m \in \mathsf{mon}(\Phi), \forall \mathcal{N} \in \mathsf{SAT}, \forall s \in \mathcal{M} \to \mathcal{N}. mst \in \Phi(\mathcal{N})\}.$$

We will prove now that $\Phi^{\supseteq}$ is monotone.

**Lemma 4.3.** *$\Phi^{\supseteq}$ is monotone, i.e., for all $\mathcal{P}, \mathcal{Q}, \in \mathsf{SAT}$, if $\mathcal{P} \subseteq \mathcal{Q}$ then $\Phi^{\supseteq}(\mathcal{P}) \subseteq \Phi^{\supseteq}(\mathcal{Q})$.*

*Proof.* Assume $\mathcal{P} \subseteq \mathcal{Q}$ and take $t \in \Phi^{\supseteq}(\mathcal{P})$. Take also $\mathcal{N} \in \mathsf{SAT}$, $m \in \mathsf{mon}(\Phi)$ and $s \in \mathcal{Q} \to \mathcal{N}$. We need to show that $mst \in \Phi(\mathcal{N})$. By Lemma 4.1, $s \in \mathcal{Q} \to \mathcal{N}$ implies $s \in \mathcal{P} \to \mathcal{N}$. The claim follows now from the assumption $t \in \Phi^{\supseteq}(\mathcal{P})$. □

Using $\Phi^{\supseteq}$ we define next another operator in an analogous way to Definition 4.9.

**Definition 4.10.** Given any $\mathcal{M} \in \mathsf{SAT}$, we define $\mathcal{I}^{\supseteq}_\mu(\mathcal{M}) = \{\mathsf{in}\, r \mid r \in \Phi^{\supseteq}(\mathcal{M})\}$ and an operator $\Psi^{\supseteq}_I : \mathsf{SAT} \to \mathsf{SAT}$ as $\Psi^{\supseteq}_I(\mathcal{M}) = \mathsf{cl}(\mathcal{I}^{\supseteq}_\mu(\mathcal{M}))$.

Clearly $\Psi^{\supseteq}_I$ is monotone, for so is $\Phi^{\supseteq}$ due to Lemma 4.3. Therefore, by the Knaster-Tarski Theorem on the complete lattice $\mathsf{SAT}$, the following definition is correct.

**Definition 4.11.** Given any operator $\Phi : \mathsf{SAT} \to \mathsf{SAT}$ we define the saturated set $\mu(\Phi)$ as follows:

$$\mu(\Phi) = \mathsf{lfp}(\Psi_I^{\supseteq}).$$

That is, for every operator $\Phi$, we define $\mu(\Phi)$ as the least fixed point of its associated monotone operator $\Psi_I^{\supseteq}$.

The following properties of the operators $\mathcal{I}_\mu, \mathcal{I}_\mu^{\supseteq}, \Psi_I$ and $\Psi_I^{\supseteq}$ will be needed later.

**Lemma 4.4.** *Let* $\mathcal{M} \in \mathsf{SAT}$. *Then*

(1) $\mathcal{I}_\mu(\mathcal{M}) \subseteq \mathsf{SN}$.
(2) $\mathcal{I}_\mu(\mathcal{M}) \subseteq \Psi_I(\mathcal{M})$.
(3) $\mathcal{I}_\mu^{\supseteq}(\mathcal{M}) \subseteq \mathsf{SN}$.
(4) $\mathcal{I}_\mu^{\supseteq}(\mathcal{M}) \subseteq \Psi_I^{\supseteq}(\mathcal{M})$.

*Proof.* For part (3) take $t \in \mathcal{I}_\mu^{\supseteq}(\mathcal{M})$, that is, $t = \mathsf{in}\, r$ for some $r \in \Phi^{\supseteq}(\mathcal{M})$. As $\Phi^{\supseteq}(\mathcal{M}) \subseteq \mathsf{SN}$ we have $r \in \mathsf{SN}$, which by definition of $\mathsf{SN}$ implies $\mathsf{in}\, r \in \mathsf{SN}$, that is, $t \in \mathsf{SN}$. To prove part (4), observe that by definition of the closure we have $\mathcal{I}_\mu^{\supseteq}(\mathcal{M}) \cap \mathsf{SN} \subseteq \Psi_I^{\supseteq}(\mathcal{M})$, but part (3) of this lemma yields $\mathcal{I}_\mu^{\supseteq}(\mathcal{M}) \cap \mathsf{SN} = \mathcal{I}_\mu^{\supseteq}(\mathcal{M})$. Parts (1) and (2) are proved analogously. □

Next we characterize the pre-fixed points of $\Psi_I$.

**Lemma 4.5.** $\Psi_I(\mathcal{M}) \subseteq \mathcal{M} \Leftrightarrow \forall t \in \Phi(\mathcal{M}).\ \mathsf{in}\, t \in \mathcal{M}$.

*Proof.* $\Rightarrow$) Assume $\Psi_I(\mathcal{M}) \subseteq \mathcal{M}$, *i.e.*, $\mathsf{cl}(\mathcal{I}_\mu(\mathcal{M})) \subseteq \mathcal{M}$. Take $t \in \Phi(\mathcal{M})$, then by definition $\mathsf{in}\, t \in \mathcal{I}_\mu(\mathcal{M})$, which, by part (2) of Lemma 4.4, implies $\mathsf{in}\, t \in \Psi_I(\mathcal{M}) \subseteq \mathcal{M}$. Therefore $\mathsf{in}\, t \in \mathcal{M}$.
$\Leftarrow$) Assume $\forall t \in \Phi(\mathcal{M}).\ \mathsf{in}\, t \in \mathcal{M}$ and take $r \in \Psi_I(\mathcal{M}) = \mathsf{cl}(\mathcal{I}_\mu(\mathcal{M}))$. The goal is $r \in \mathcal{M}$. As $\mathcal{M} \in \mathsf{SAT}$ it suffices to show $\mathcal{I}_\mu(\mathcal{M}) \cap \mathsf{SN} \subseteq \mathcal{M}$, thus the goal will follow by the minimality of the closure. By Lemma 4.4, part (1), we have $\mathcal{I}_\mu(\mathcal{M}) \subseteq \mathsf{SN}$. Hence we only have to show $\mathcal{I}_\mu(\mathcal{M}) \subseteq \mathcal{M}$. Take $\mathsf{in}\, t \in \mathcal{I}_\mu(\mathcal{M})$, so $t \in \Phi(\mathcal{M})$ which by assumption implies $\mathsf{in}\, t \in \mathcal{M}$. Therefore $\mathcal{I}_\mu(\mathcal{M}) \subseteq \mathcal{M}$. □

The soundness of the typing rule $(\mu I)$ depends on the following:

**Lemma 4.6.** $\mu(\Phi)$ *is a pre-fixed point of* $\Psi_I$. *i.e.,* $\Psi_I\big(\mu(\Phi)\big) \subseteq \mu(\Phi)$.

*Proof.* As $\mu(\Phi)$ is a fixed-point of $\Psi_I^{\supseteq}$, it suffices to prove that $\Psi_I\Big(\Psi_I^{\supseteq}\big(\mu(\Phi)\big)\Big) \subseteq \Psi_I^{\supseteq}\big(\mu(\Phi)\big)$. To show this, we use Lemma 4.5. Take $t \in \Phi\Big(\Psi_I^{\supseteq}\big(\mu(\Phi)\big)\Big)$, therefore $\mathsf{in}\, t \in \mathcal{I}_\mu^{\supseteq}\Big(\Psi_I^{\supseteq}\big(\mu(\Phi)\big)\Big) \subseteq \Psi_I^{\supseteq}\Big(\Psi_I^{\supseteq}\big(\mu(\Phi)\big)\Big)$, the last inclusion concluded by Lemma 4.4, part (4). Therefore, as $\mu(\Phi)$ is a fixed-point of $\Psi_I^{\supseteq}$, we conclude $\mathsf{in}\, t \in \Psi_I^{\supseteq}\big(\mu(\Phi)\big)$. □

Next, we built saturated sets $\mathcal{E}_\mu(\mathcal{M})$ necessary to prove the soundness of the typing rules $(\mu E)$, $(\mu E^+)$, $(\mu E^i)$.

**Definition 4.12.** Given $\Phi : \mathsf{SAT} \to \mathsf{SAT}$ and $\mathcal{M} \in \mathsf{SAT}$ we define

$$\mathcal{E}_\mu(\mathcal{M}) = \Big\{ r \in \mathsf{SN} \;\Big|\; \begin{array}{l} \forall m \in \mathsf{mon}(\Phi).\, \forall \mathcal{N} \in \mathsf{SAT}. \\ \big(\forall s \in \Phi(\mathcal{N}) \to \mathcal{N}.\, \mathsf{It}(m,s,r) \in \mathcal{N}\big) \wedge \\ \big(\forall s \in \Phi(\mathcal{M} \times \mathcal{N}) \to \mathcal{N}.\, \mathsf{Rec}(m,s,r) \in \mathcal{N}\big) \wedge \\ \mathsf{in}^{-1}(m,r) \in \Phi(\mathcal{M}) \Big\} \end{array}$$

and $\Psi_E : \mathsf{SAT} \to \mathsf{SAT}$ as

$$\Psi_E(\mathcal{M}) = \mathsf{cl}(\mathcal{E}_\mu(\mathcal{M})).$$

The next properties will be required later.

**Lemma 4.7.** *Let $\mathcal{M} \in \mathsf{SAT}$. Then*

(1) $\mathcal{E}_\mu(\mathcal{M}) \in \mathsf{SAT}$.
(2) $\mathcal{E}_\mu(\mathcal{M}) = \Psi_E(\mathcal{M})$.

*Proof.*

(1) It is clear that $\mathcal{E}_\mu(\mathcal{M}) \subseteq \mathsf{SN}$.
Take $E[x] \in \mathsf{SN}$. We have to show that $E[x] \in \mathcal{E}_\mu(\mathcal{M})$. Fix $m \in \mathsf{mon}(\Phi)$ and $\mathcal{N} \in \mathsf{SAT}$.
- Assume $s \in \Phi(\mathcal{N}) \to \mathcal{N}$.
The goal is $\mathsf{It}(m,s,E[x]) \in \mathcal{N}$. Observe that this term is again an evaluation context, say $E'[x]$. As $\mathcal{N} \in \mathsf{SAT}$ it suffices to show that $E'[x] \in \mathsf{SN}$. We have $E[x] \in \mathsf{SN}$ and $s \in \Phi(\mathcal{N}) \to \mathcal{N} \subseteq \mathsf{SN}$, which implies $s \in \mathsf{SN}$. Similarly $m \in \mathsf{mon}(\Phi) \subseteq \mathsf{SN}$. Hence all $m, s, E[x] \in \mathsf{SN}$ which by properties of $\mathsf{SN}$ implies $\mathsf{It}(m,s,E[x]) \in \mathsf{SN}$.
- Assume $s \in \Phi(\mathcal{M} \times \mathcal{N}) \to \mathcal{N}$. The goal is $\mathsf{Rec}(m,s,E[x]) \in \mathcal{N}$. As in the previous case we obtain $m, s \in \mathsf{SN}$, therefore by properties of $\mathsf{SN}$ we conclude $E'[x] = \mathsf{Rec}(m,s,E[x]) \in \mathsf{SN}$. Therefore, as $\mathcal{N} \in \mathsf{SAT}$ we get $E'[x] \in \mathcal{N}$.
- The goal is $\mathsf{in}^{-1}(m,E[x]) \in \Phi(\mathcal{M})$. Again we have $m \in \mathsf{SN}$ and, as $E[x] \in \mathsf{SN}$ by properties of $\mathsf{SN}$ we get $E'[x] = \mathsf{in}^{-1}(m,E[x]) \in \mathsf{SN}$, which yields $E'[x] \in \Phi(\mathcal{M})$, for $\Phi(\mathcal{M}) \in \mathsf{SAT}$.

The other closure rules for $\mathsf{SAT}$ sets are proved in a similar way.

(2) First observe that $\mathcal{E}_\mu(\mathcal{M}) = \mathcal{E}_\mu(\mathcal{M}) \cap \mathsf{SN} \subseteq \mathsf{cl}(\mathcal{E}_\mu(\mathcal{M})) = \Psi_E(\mathcal{M})$. For the reverse inclusion, observe that by part (1) of this lemma we have $\mathcal{E}_\mu(\mathcal{M}) \in \mathsf{SAT}$. Therefore, by minimality of the closure, we get $\Psi_E(\mathcal{M}) = \mathsf{cl}(\mathcal{E}_\mu(\mathcal{M})) \subseteq \mathcal{E}_\mu(\mathcal{M})$. □

Next, we characterize the post-fixed points of $\Psi_E$.

**Lemma 4.8.**

$$\mathcal{M} \subseteq \Psi_E(\mathcal{M}) \Leftrightarrow \begin{array}{l} \forall r \in \mathcal{M}. \forall m \in \mathsf{mon}(\Phi).\, \forall \mathcal{N} \in \mathsf{SAT}. \\ \big(\forall s \in \Phi(\mathcal{N}) \to \mathcal{N}.\, \mathsf{It}(m,s,r) \in \mathcal{N}\big) \wedge \\ \big(\forall s \in \Phi(\mathcal{M} \times \mathcal{N}) \to \mathcal{N}.\, \mathsf{Rec}(m,s,r) \in \mathcal{N}\big) \wedge \\ \mathsf{in}^{-1}(m,r) \in \Phi(\mathcal{M}). \end{array}$$

*Proof.* Call $\Box(r)$ to the condition on the right hand side for a given $r \in \mathcal{M}$.
$\Rightarrow$) Assume $\mathcal{M} \subseteq \Psi_E(\mathcal{M})$. We have to show $\Box(r)$ for all $r \in \mathcal{M}$. Take $r \in \mathcal{M}$, by part (2) of Lemma 4.7 we have $\mathcal{M} \subseteq \mathcal{E}_\mu(\mathcal{M})$ and observing that $\mathcal{E}_\mu(\mathcal{M}) = \{r \in \mathsf{SN} \mid \Box(r)\}$ we yield $\Box(r)$ as desired.
$\Leftarrow$) Assume $\forall r \in \mathcal{M}.\Box(r)$ and take $r \in \mathcal{M}$. We have to show that $r \in \Psi_E(M)$. By part (2) of Lemma 4.7 it suffices to show that $r \in \mathcal{E}_\mu(\mathcal{M})$. We have $r \in \mathsf{SN}$, for $\mathcal{M} \subseteq \mathsf{SN}$. Moreover $\Box(r)$ holds by assumption, which implies $r \in \mathcal{E}_\mu(\mathcal{M})$. □

The following lemma will ensure the soundness of the typing rules for elimination $(\mu E), (\mu E^+), (\mu E^i)$.

**Lemma 4.9.** *$\mu(\Phi)$ is a post-fixed point of $\Psi_E$. i.e., $\mu(\Phi) \subseteq \Psi_E(\mu(\Phi))$.*

*Proof.* Our goal is $\mu(\Phi) \subseteq \Psi_E\big(\mu(\Phi)\big)$. To prove this, we will use the principle of extended induction on $\mu(\Phi)$ given in Corollary 4.1 of page 732. Therefore the goal becomes $\Psi_I^{\supseteq}\Big(\mu(\Phi) \cap \Psi_E\big(\mu(\Phi)\big)\Big) \subseteq \Psi_E\big(\mu(\Phi)\big)$.
Set $\mathcal{L} = \mu(\Phi)$, $\mathcal{L}' = \mathcal{L} \cap \Psi_E(\mathcal{L})$. The goal is $\Psi_I^{\supseteq}(\mathcal{L}') \subseteq \Psi_E(\mathcal{L})$. By monotonicity of the closure it suffices to show $\mathcal{I}_\mu^{\supseteq}(\mathcal{L}') \subseteq \mathcal{E}_\mu(\mathcal{L})$. Take $t \in \mathcal{I}_\mu^{\supseteq}(\mathcal{L}')$, that is, $t = \mathsf{in}\, r$ with $r \in \Phi^{\supseteq}(\mathcal{L}')$. We need to show $\mathsf{in}\, r \in \mathcal{E}_\mu(\mathcal{L})$.
First observe that $\mathsf{in}\, r \in \mathsf{SN}$, for $r \in \Phi^{\supseteq}(\mathcal{L}') \subseteq \mathsf{SN}$ and by the properties of $\mathsf{SN}$. Next we have to prove that $\Box(\mathsf{in}\, r)$ (*cf.* proof of Lem. 4.8). To this purpose let us fix $m \in \mathsf{mon}(\Phi)$ and $\mathcal{N} \in \mathsf{SAT}$.

- Take $s \in \Phi(\mathcal{N}) \to \mathcal{N}$. We want to show that $\mathsf{It}(m, s, \mathsf{in}\, r) \in \mathcal{N}$. Using that $\mathcal{N} \in \mathsf{SAT}$, it suffices to show that $s\big(m(\lambda x.\mathsf{It}(m, s, x))r\big) \in \mathcal{N}$. As $s \in \Phi(\mathcal{N}) \to \mathcal{N}$ we only have to show $m(\lambda x.\mathsf{It}(m, s, x))r \in \Phi(\mathcal{N})$ but observing that $r \in \Phi^{\supseteq}(\mathcal{L}')$ it suffices to prove that $m \in \mathsf{mon}(\Phi)$, $\mathcal{N} \in \mathsf{SAT}$ and $\lambda x.\mathsf{It}(m, s, x) \in \mathcal{L}' \to \mathcal{N}$. The first two claims are given, and to prove the last one we will show that $\mathsf{It}(m, s, x) \in \mathsf{S}_x(\mathcal{L}', \mathcal{N})$. Take $q \in \mathcal{L}'$, we prove $\mathsf{It}(m, s, x)[x := q] \in \mathcal{N}$, which, as w.l.o.g. $x \notin FV(m, s)$, equals $\mathsf{It}(m, s, q) \in \mathcal{N}$. We have $\mathcal{L}' \subseteq \Psi_E(\mathcal{L}) = \mathcal{E}_\mu(\mathcal{L})$, the equality given by part (2) of Lemma 4.7. Therefore $q \in \mathcal{E}_\mu(\mathcal{L})$ which immediately yields the needed $\mathsf{It}(m, s, q) \in \mathcal{N}$.
- Take $s \in \Phi(\mathcal{L} \times \mathcal{N}) \to \mathcal{N}$. We need to prove $\mathsf{Rec}(m, s, \mathsf{in}\, r) \in \mathcal{N}$. By reasoning as in the previous case we only have to show that $\lambda z.\langle(\lambda yy)z, (\lambda x.\mathsf{Rec}(m, s, x))z\rangle \in \mathcal{L}' \to \mathcal{L} \times \mathcal{N}$. It suffices to prove $\big\langle(\lambda yy)z, (\lambda x.\mathsf{Rec}(m, s, x))z\big\rangle \in \mathsf{S}_z(\mathcal{L}', \mathcal{L} \times \mathcal{N})$, so we take $q \in \mathcal{L}'$ and show that $\langle(\lambda yy)q, (\lambda x.\mathsf{Rec}(m, s, x))q\rangle \in \mathcal{L} \times \mathcal{N}$. To this purpose we prove two things:
  - $(\lambda yy)q \in \mathcal{L}$. Clearly we have $\lambda yy \in \mathcal{L} \to \mathcal{L}$ and as $q \in \mathcal{L}' \subseteq \mathcal{L}$ we get $(\lambda yy)q \in \mathcal{L}$.
  - $(\lambda x.\mathsf{Rec}(m, s, x))q \in \mathcal{N}$. It suffices to show $\lambda x.\mathsf{Rec}(m, s, x) \in \mathcal{L}' \to \mathcal{N}$, that is, $\mathsf{Rec}(m, s, x) \in \mathsf{S}_x(\mathcal{L}', \mathcal{N})$. Take $p \in \mathcal{L}'$, we will show $\mathsf{Rec}(m, s, x)[x := p] \in \mathcal{N}$, where w.l.o.g. $x \notin FV(m, s)$, which implies to prove $\mathsf{Rec}(m, s, p) \in \mathcal{N}$. We have $\mathcal{L}' \subseteq \Psi_E(\mathcal{L}) = \mathcal{E}_\mu(\mathcal{L})$, the equality given by part (2) of Lemma 4.7. Therefore $p \in \mathcal{E}_\mu(\mathcal{L})$ which immediately yields $\mathsf{Rec}(m, s, p) \in \mathcal{N}$.

- The goal is to prove $\mathsf{in}^{-1}(m, \mathsf{in}\, r) \in \Phi(\mathcal{L})$. As $r \in \Phi^{\supseteq}(\mathcal{L}')$ and $m \in \mathsf{mon}(\Phi)$ it suffices to show $\lambda zz \in \mathcal{L}' \to \mathcal{L}$, that is, $z \in \mathsf{S}_z(\mathcal{L}', \mathcal{L})$. Then we take $s \in \mathcal{L}'$ and want to show $s \in \mathcal{L}$, but this is immediate from $\mathcal{L}' \subseteq \mathcal{L}$.

Therefore, $\Box(\mathsf{in}\, r)$. □

Our final constructions of saturated sets will be for coinductive types.

4.3.2. *Saturated sets for coinductive types*

In this section we give saturated sets corresponding to coinductive types. These constructions are obtained by a straightforward, yet not trivial, dualization of the constructions for inductive types given in the previous section and are, to our knowledge, new.

Let us start with the construction needed to prove the soundness of the introduction typing rules $(\nu I), (\nu I^+), (\nu I^i)$.

**Definition 4.13.** Given $\Phi : \mathsf{SAT} \to \mathsf{SAT}$ and $\mathcal{M} \in \mathsf{SAT}$, we define

$$\begin{array}{rcl} \mathcal{I}_\nu(\mathcal{M}) & = & \{\mathsf{CoIt}(m,s,t) \mid m \in \mathsf{mon}(\Phi),\ s \in \mathcal{N} \to \Phi(\mathcal{N}),\ t \in \mathcal{N}, \mathcal{N} \in \mathsf{SAT}\} \\ & \cup & \{\mathsf{CoRec}(m,s,t) \mid m \in \mathsf{mon}(\Phi),\ s \in \mathcal{N} \to \Phi(\mathcal{M}+\mathcal{N}),\ t \in \mathcal{N} \in \mathsf{SAT}\} \\ & \cup & \{\mathsf{out}^{-\mathbf{1}}(m,t) \mid m \in \mathsf{mon}(\Phi),\ t \in \Phi(\mathcal{M})\} \end{array}$$

and $\Theta_I : \mathsf{SAT} \to \mathsf{SAT}$ such that $\Theta_I(\mathcal{M}) = \mathsf{cl}(\mathcal{I}_\nu(\mathcal{M}))$.

The following lemma will be useful later

**Lemma 4.10.** *Let $\mathcal{M} \in \mathsf{SAT}$. Then*

(1) $\mathcal{I}_\nu(\mathcal{M}) \subseteq \mathsf{SN}$.
(2) $\mathcal{I}_\nu(\mathcal{M}) \subseteq \Theta_I(\mathcal{M})$.

*Proof.*

(1) Take $r \in \mathcal{I}_\nu(\mathcal{M})$. We have three cases: if $r = \mathsf{CoIt}(m,s,t)$ then we have $m, s, t \in \mathsf{SN}$, for they belong to some saturated set. Therefore, by properties of $\mathsf{SN}$ we also have $\mathsf{CoIt}(m,s,t) \in \mathsf{SN}$. The cases $r = \mathsf{CoRec}(m,s,t)$ and $r = \mathsf{out}^{-1}(m,t)$ are similar.
(2) By definition of closure we have $\mathcal{I}_\nu(\mathcal{M}) \cap \mathsf{SN} \subseteq \mathsf{cl}(\mathcal{I}_\nu(\mathcal{M}))$ which, by part (1) of this lemma is equivalent to $\mathcal{I}_\nu(\mathcal{M}) \subseteq \mathsf{cl}(\mathcal{I}_\nu(\mathcal{M})) = \Theta_I(\mathcal{M})$. □

We characterize now the pre-fixed points of $\Theta_I$.

**Lemma 4.11.**

$$\begin{array}{rl} \Theta_I(\mathcal{M}) \subseteq \mathcal{M} \Leftrightarrow & \forall m \in \mathsf{mon}(\Phi).\, \forall \mathcal{N} \in \mathsf{SAT}. \\ & \big(\forall t \in \mathcal{N}\ \forall s \in \mathcal{N} \to \Phi(\mathcal{N}).\ \mathsf{CoIt}(m,s,t) \in \mathcal{M}\big) \wedge \\ & \big(\forall t \in \mathcal{N}\ \forall s \in \mathcal{N} \to \Phi(\mathcal{M}+\mathcal{N}).\ \mathsf{CoRec}(m,s,t) \in \mathcal{M}\big) \wedge \\ & \big(\forall t \in \Phi(\mathcal{M}).\, \mathsf{out}^{-\mathbf{1}}(m,t) \in \mathcal{M}\big). \end{array}$$

*Proof.* $\Rightarrow$) Assume $\Theta_I(\mathcal{M}) \subseteq \mathcal{M}$. By part (2) of Lemma 4.10 we get $\mathcal{I}_\nu(\mathcal{M}) \subseteq \mathcal{M}$.

Take $m \in \mathsf{mon}(\Phi)$ and $\mathcal{N} \in \mathsf{SAT}$. We prove every part of the conjunction: If $t \in \mathcal{N}$ and $s \in \mathcal{N} \to \Phi(\mathcal{N})$ then we get $\mathsf{Colt}(m,s,t) \in \mathcal{I}_\nu(\mathcal{M})$, which implies $\mathsf{Colt}(m,s,t) \in \mathcal{M}$. Similarly, if $t \in \mathcal{N}$ and $s \in \mathcal{N} \to \Phi(\mathcal{M}+\mathcal{N})$ then $\mathsf{CoRec}(m,s,t) \in \mathcal{I}_\nu(\mathcal{M}) \subseteq \mathcal{M}$. Finally $t \in \Phi(\mathcal{M})$ yields $\mathsf{out}^{-1}(m,t) \in \mathcal{I}_\nu(\mathcal{M}) \subseteq \mathcal{M}$.

$\Leftarrow$) Assume the condition on the right hand side. We have $\Theta_I(\mathcal{M}) = \mathsf{cl}(\mathcal{I}_\nu(\mathcal{M}))$. By minimality of the closure it suffices to show $\mathcal{I}_\nu(\mathcal{M}) \cap \mathsf{SN} \subseteq \mathcal{M}$ but by part (1) of Lemma 4.10 this is equivalent to $\mathcal{I}_\nu(\mathcal{M}) \subseteq \mathcal{M}$, which follows immediately from the assumption and the definition of $\mathcal{I}_\nu(\mathcal{M})$. □

Next, we develop the tools needed to prove the soundness of the typing rule $(\nu E)$.

**Definition 4.14.** Given $\Phi : \mathsf{SAT} \to \mathsf{SAT}$ and $\mathcal{M} \in \mathsf{SAT}$ we define $\mathcal{E}_\nu(\mathcal{M}) = \{r \in \mathsf{SN} \mid \mathsf{out}\, r \in \Phi(\mathcal{M})\}$ and the operator $\Theta_E : \mathsf{SAT} \to \mathsf{SAT}$ with $\Theta_E(\mathcal{M}) = \mathsf{cl}(\mathcal{E}_\nu(\mathcal{M}))$.

Our goal is to associate any operator $\Phi$ with a saturated set $\nu(\Phi)$ obtained as the greatest fixed point of a monotone operator related to $\Phi$. As we do not want to restrict ourselves to a monotone $\Phi$, we proceed in a similar way to the construction of the set $\mu(\Phi)$ discussed in the previous section.

Let us define $\Phi^{\subseteq} : \mathsf{SAT} \to \mathsf{SAT}$ as $\Phi^{\subseteq}(\mathcal{M}) = \mathsf{cl}(\mathsf{A}(\mathcal{M}))$ with

$$\mathsf{A}(\mathcal{M}) = \{mqr \mid m \in \mathsf{mon}(\Phi),\ q \in \mathcal{N} \to \mathcal{M},\ r \in \Phi(\mathcal{N}) \text{ for some } \mathcal{N} \in \mathsf{SAT}\}.$$

We prove now that $\Phi^{\subseteq}$ is monotone.

**Lemma 4.12.** *For all $\mathcal{P}, \mathcal{Q} \in \mathsf{SAT}$, if $\mathcal{P} \subseteq \mathcal{Q}$ then $\Phi^{\subseteq}(\mathcal{P}) \subseteq \Phi^{\subseteq}(\mathcal{Q})$, that is, $\Phi^{\subseteq}$ is monotone.*

*Proof.* Assume $\mathcal{P} \subseteq \mathcal{Q}$. Take $mqr \in \Phi^{\subseteq}(\mathcal{P})$, then $m \in \mathsf{mon}(\Phi)$, $q \in \mathcal{N} \to \mathcal{P}$, $r \in \Phi(\mathcal{N})$. $q \in \mathcal{N} \to \mathcal{P}$ implies, by Lemma 4.2, $q \in \mathcal{N} \to \mathcal{Q}$. Therefore we have $mqr \in \Phi^{\subseteq}(\mathcal{Q})$. □

The following properties related to $\mathsf{A}(\mathcal{M})$ will be required later.

**Lemma 4.13.** *Let $\mathcal{M} \in \mathsf{SAT}$. Then*

(1) $\mathsf{A}(\mathcal{M}) \subseteq \Phi(\mathcal{M})$.
(2) $\mathsf{A}(\mathcal{M}) \subseteq \mathsf{SN}$.
(3) $\mathsf{A}(\mathcal{M}) \subseteq \Phi^{\subseteq}(\mathcal{M})$.
(4) $\Phi^{\subseteq}(\mathcal{M}) \subseteq \Phi(\mathcal{M})$.

*Proof.*

(1) Take $t \in \mathsf{A}(\mathcal{M})$, *i.e.*, $t = mqr$ with $m \in \mathsf{mon}(\Phi)$, $q \in \mathcal{N} \to \mathcal{M}$, $r \in \Phi(\mathcal{N})$ for some $\mathcal{N} \in \mathsf{SAT}$. $m \in \mathsf{mon}(\Phi) \Rightarrow m \in (\mathcal{N} \to \mathcal{M}) \to (\Phi(\mathcal{N}) \to \Phi(\mathcal{M})) \Rightarrow mq \in \Phi(\mathcal{N}) \to \Phi(\mathcal{M}) \Rightarrow mqr \in \Phi(\mathcal{M})$, *i.e.* $t \in \Phi(\mathcal{M})$.
(2) $\mathsf{A}(\mathcal{M}) \subseteq \Phi(\mathcal{M}) \subseteq \mathsf{SN}$.

(3) $\mathsf{A}(\mathcal{M}) = \mathsf{A}(\mathcal{M}) \cap \mathsf{SN} \subseteq \mathsf{cl}(\mathsf{A}(\mathcal{M})) = \Phi^{\subseteq}(\mathcal{M})$.
(4) As $\Phi(\mathcal{M}) \in \mathsf{SAT}$, by minimality of the closure it suffices to show $\mathsf{A}(\mathcal{M}) \cap \mathsf{SN} \subseteq \Phi(\mathcal{M})$, but by part (2) of this lemma this is equivalent to $\mathsf{A}(\mathcal{M}) \subseteq \Phi(\mathcal{M})$, which was proved in part (1) of this lemma. □

Using $\Phi^{\subseteq}$ we define an operator $\Theta^{\subseteq}_E$ in an analogous way to Definition 4.14.

**Definition 4.15.** Given $\Phi : \mathsf{SAT} \to \mathsf{SAT}$ and $\mathcal{M} \in \mathsf{SAT}$, we define $\mathcal{E}^{\subseteq}_\nu(\mathcal{M}) = \{r \in \mathsf{SN} \mid \mathsf{out}\, r \in \Phi^{\subseteq}(\mathcal{M})\}$ and the operator $\Theta^{\subseteq}_E : \mathsf{SAT} \to \mathsf{SAT}$ as $\Theta^{\subseteq}_E(\mathcal{M}) = \mathsf{cl}(\mathcal{E}^{\subseteq}_\nu(\mathcal{M}))$.

The following properties of $\mathcal{E}_\nu(\mathcal{M}), \mathcal{E}^{\subseteq}_\nu(\mathcal{M})$ will be needed later.

**Lemma 4.14.** *Let* $\mathcal{M} \in \mathsf{SAT}$. *Then*

(1) $\mathcal{E}_\nu(\mathcal{M}) \in \mathsf{SAT}$.
(2) $\mathcal{E}^{\subseteq}_\nu(\mathcal{M}) \in \mathsf{SAT}$.
(3) $\mathcal{E}_\nu(\mathcal{M}) = \Theta_E(\mathcal{M})$.
(4) $\mathcal{E}^{\subseteq}_\nu(\mathcal{M}) = \Theta^{\subseteq}_E(\mathcal{M})$.

*Proof.*

(1) Clearly we have $\mathcal{E}_\nu(\mathcal{M}) \subseteq \mathsf{SN}$. We need to prove all rules for saturated sets. As an example take $E[x] \in \mathsf{SN}$. The goal is $E[x] \in \mathcal{E}_\nu(\mathcal{M})$, *i.e.*, $\mathsf{out}\, E[x] \in \Phi(\mathcal{M})$. By properties of $\mathsf{SN}$, $E[x] \in \mathsf{SN}$ implies $\mathsf{out}\, E[x] \in \mathsf{SN}$, but $\mathsf{out}\, E[x]$ is an evaluation context with $\bullet$ filled by $x$, say $E'[x] \in \mathsf{SN}$. Therefore $\Phi(\mathcal{M}) \in \mathsf{SAT}$ implies $E'[x] \in \Phi(\mathcal{M})$. The remaining rules are easily proved.
(2) Analogous to part (1).
(3) $\subseteq$) We have $\mathcal{E}_\nu(\mathcal{M}) \cap \mathsf{SN} \subseteq \mathsf{cl}(\mathcal{E}_\nu(\mathcal{M}))$, which, as $\mathcal{E}_\nu(\mathcal{M}) \subseteq \mathsf{SN}$, is equivalent to $\mathcal{E}_\nu(\mathcal{M}) \subseteq \mathsf{cl}(\mathcal{E}_\nu(\mathcal{M})) = \Theta_E(\mathcal{M})$.
$\supseteq$) By part (1) of this lemma, using the minimality of the closure we have $\Theta_E(\mathcal{M}) = \mathsf{cl}(\mathcal{E}_\nu(\mathcal{M})) \subseteq \mathcal{E}_\nu(\mathcal{M})$.
(4) Analogous to part (3). □

We characterize now the post-fixed points of $\Theta_E(\mathcal{M})$.

**Lemma 4.15.** $\mathcal{M} \subseteq \Theta_E(\mathcal{M}) \Leftrightarrow \forall t \in \mathcal{M}.\, \mathsf{out}\, t \in \Phi(\mathcal{M})$

*Proof.* $\Rightarrow$) Let $t \in \mathcal{M}$. By assumption we get $t \in \Theta_E(\mathcal{M})$, and by part (3) of Lemma 4.14, $t \in \mathcal{E}_\nu(\mathcal{M})$, which by definition of $\mathcal{E}_\nu(\mathcal{M})$ yields $\mathsf{out}\, t \in \Phi(\mathcal{M})$.
$\Leftarrow$) Take $t \in \mathcal{M}$, by assumption we get $\mathsf{out}\, t \in \Phi(\mathcal{M})$. On the other hand, as $\mathcal{M} \subseteq \mathsf{SN}$, we get $t \in \mathsf{SN}$. Therefore $t \in \mathcal{E}_\nu(\mathcal{M})$, which by part (3) of Lemma 4.14, is the same as $t \in \Theta_E(\mathcal{M})$. □

We can finally define a saturated set corresponding to coinductive types. The monotonicity of $\Phi^{\subseteq}$ given by Lemma 4.12, implies that $\Theta^{\subseteq}_E$ is monotone. Therefore by the Knaster-Tarski theorem the following definition is correct.

**Definition 4.16.** Given any operator $\Phi : \mathsf{SAT} \to \mathsf{SAT}$ we define the saturated set $\nu(\Phi)$ as follows:

$$\nu(\Phi) = \mathsf{gfp}(\Theta^{\subseteq}_{\overline{E}}).$$

That is, we define $\nu(\Phi)$ as the greatest fixed point of its associated monotone operator $\Theta^{\subseteq}_{\overline{E}}$.

The following properties of $\nu(\Phi)$ will be relevant for the proof of soundness.

**Lemma 4.16.** *$\nu(\Phi)$ is a post-fixed point of $\Theta_E$. i.e., $\nu(\Phi) \subseteq \Theta_E(\nu(\Phi))$.*

*Proof.* By Lemma 4.15 it suffices to show $\forall t \in \nu(\Phi).\, \mathsf{out}\, t \in \Phi\big(\nu(\Phi)\big)$. If $t \in \nu(\Phi) = \Theta^{\subseteq}_{\overline{E}}\big(\nu(\Phi)\big)$ then by part (4) of Lemma 4.14, $t \in \mathcal{E}^{\subseteq}_{\nu}\big(\nu(\Phi)\big)$, which implies $\mathsf{out}\, t \in \Phi^{\subseteq}\big(\nu(\Phi)\big)$. Finally part (4) of Lemma 4.13 yields $\mathsf{out}\, t \in \Phi\big(\nu(\Phi)\big)$. □

**Lemma 4.17.** *$\nu(\Phi)$ is a pre-fixed point of $\Theta_I$. i.e., $\Theta_I(\nu(\Phi)) \subseteq \nu(\Phi)$*

*Proof.* The proof is by means of the principle of extended coinduction given in Corollary 4.1. Hence, the goal becomes

$$\Theta_I\big(\nu(\Phi)\big) \subseteq \Theta^{\subseteq}_{\overline{E}}\Big(\nu(\Phi) \cup \Theta_I\big(\nu(\Phi)\big)\Big).$$

Set $\mathcal{G} = \nu(\Phi)$, $\mathcal{G}' = \mathcal{G} \cup \Theta_I(\mathcal{G})$. The goal is $\Theta_I(\mathcal{G}) \subseteq \Theta^{\subseteq}_{\overline{E}}(\mathcal{G}')$. By monotonicity of the closure it suffices to show $\mathcal{I}_\nu(\mathcal{G}) \subseteq \mathcal{E}^{\subseteq}_{\nu}(\mathcal{G}')$. Take $r \in \mathcal{I}_\nu(\mathcal{G})$, to show $r \in \mathcal{E}^{\subseteq}_{\nu}(\mathcal{G}')$ it suffices to show $\mathsf{out}\, r \in \Phi^{\subseteq}(\mathcal{G}')$ (we have $r \in \mathsf{SN}$, for Lem. 4.10 yields $\mathcal{I}_\nu(\mathcal{G}) \subseteq \mathsf{SN}$). Let us analyze three cases according to the definition of $\mathcal{I}_\nu(\mathcal{G})$:

- $r = \mathsf{CoIt}(m, s, t)$ with $m \in \mathsf{mon}(\Phi), s \in \mathcal{N} \to \Phi(\mathcal{N})$ and $t \in \mathcal{N}$. By properties of saturated sets it suffices to show that $m\big(\lambda z.\mathsf{CoIt}(m, s, z)\big)(st) \in \Phi^{\subseteq}(\mathcal{G}')$ and using part (3) of Lemma 4.13 we prove only that $m\big(\lambda z.\mathsf{CoIt}(m, s, z)\big)(st) \in \mathsf{A}(\mathcal{G}')$. We have by assumption $m \in \mathsf{mon}(\Phi)$ and clearly $st \in \Phi(\mathcal{N})$. To prove $\lambda z.\mathsf{CoIt}(m, s, z) \in \mathcal{N} \to \mathcal{G}'$, we show $\mathsf{CoIt}(m, s, z) \in \mathsf{S}_z(\mathcal{N}, \mathcal{G}')$. Taking $q \in \mathcal{N}$ we show $\mathsf{CoIt}(m, s, z)[z := q] = \mathsf{CoIt}(m, s, q) \in \mathcal{G}'$ (w.l.o.g. $z \notin FV(m, s)$). Finally $\mathsf{CoIt}(m, s, q) \in \mathcal{I}_\nu(\mathcal{G})$ implies, by part (2) of Lemma 4.10, that $\mathsf{CoIt}(m, s, q) \in \Theta_I(\mathcal{G}) \subseteq \mathcal{G}'$.
- $r = \mathsf{CoRec}(m, s, t)$ with $m \in \mathsf{mon}(\Phi), s \in \mathcal{N} \to \Phi(\mathcal{G} + \mathcal{N})$ and $t \in \mathcal{N}$. By a similar reasoning as in the previous case we only need to show that $m\big([\mathsf{Id}, \lambda z.\mathsf{CoRec}(m, s, z)]\big)(st) \in \mathsf{A}(\mathcal{G}')$. We have $m \in \mathsf{mon}(\Phi)$ and clearly $st \in \Phi(\mathcal{G} + \mathcal{N})$. It remains to show that $[\mathsf{Id}, \lambda z.\mathsf{CoRec}(m, s, z)] \in \mathcal{G} + \mathcal{N} \to \mathcal{G}'$. Recalling that $[\mathsf{Id}, \lambda z.\mathsf{CoRec}(m, s, z)] = \lambda x.\mathsf{case}(x, y.y, z.\mathsf{CoRec}(m, s, z))$ the goal reduces to show $\mathsf{case}(x, y.y, z.\mathsf{CoRec}(m, s, z)) \in \mathsf{S}_x(\mathcal{G} + \mathcal{N}, \mathcal{G}')$. Therefore we take $q \in \mathcal{G} + \mathcal{N}$ and prove that $\mathsf{case}(x, y.y, z.\mathsf{CoRec}(m, s, z)) \in \mathcal{G}'$, which, by properties of saturated sets, reduces to the next two claims:
  - $y \in \mathsf{S}_y(\mathcal{G}, \mathcal{G}')$. This holds trivially for $\mathcal{G} \subseteq \mathcal{G}'$.
  - $\mathsf{CoRec}(m, s, z) \in \mathsf{S}_z(\mathcal{N}, \mathcal{G}')$. Take $p \in \mathcal{N}$ and show $\mathsf{CoRec}(m, s, z)[z := p] = \mathsf{CoRec}(m, s, p) \in \mathcal{G}'$ (w.l.o.g. $z \notin FV(m, s)$). Finally observe that $\mathsf{CoRec}(m, s, p) \in \mathcal{I}_\nu(\mathcal{G})$ and part (2) of Lemma 4.10 yields $\mathsf{CoRec}(m, s, p) \in \Theta_I(\mathcal{G}) \subseteq \mathcal{G}'$.

- $r = \mathsf{out}^{-1}(m,t)$ with $m \in \mathsf{mon}(\Phi)$ and $t \in \Phi(\mathcal{G})$. Again we only need to show that $m(\lambda zz)t \in \mathsf{A}(\mathcal{G}')$. Having $m \in \mathsf{mon}(\Phi)$ and $t \in \Phi(\mathcal{G})$, it only remains to show that $\lambda zz \in \mathcal{G} \to \mathcal{G}'$, which is immediate consequence of $\mathcal{G} \subseteq \mathcal{G}'$. □

We are ready to prove the soundness of all typing rules involving (co)inductive types.

**Proposition 4.4** (soundness of the constructions)**.** *Given* $\Phi : \mathsf{SAT} \to \mathsf{SAT}$ *the following holds.*

(1) *If* $t \in \Phi(\mu(\Phi))$, *then* $\mathsf{in}\, t \in \mu(\Phi)$.
(2) *If* $r \in \mu(\Phi), m \in \mathsf{mon}(\Phi), \mathcal{N} \in \mathsf{SAT}$ *and* $s \in \Phi(\mathcal{N}) \to \mathcal{N}$, *then* $\mathsf{It}(m,s,r) \in \mathcal{N}$.
(3) *If* $r \in \mu(\Phi), m \in \mathsf{mon}(\Phi), \mathcal{N} \in \mathsf{SAT}$ *and* $s \in \Phi(\mu(\Phi) \times \mathcal{N}) \to \mathcal{N}$, *then* $\mathsf{Rec}(m,s,r) \in \mathcal{N}$.
(4) *If* $m \in \mathsf{mon}(\Phi)$ *and* $r \in \mu(\Phi)$, *then* $\mathsf{in}^{-1}(m,r) \in \Phi(\mu(\Phi))$.
(5) *If* $t \in \nu(\Phi)$, *then* $\mathsf{out}\, t \in \Phi(\nu(\Phi))$.
(6) *If* $\mathcal{N} \in \mathsf{SAT}, r \in \mathcal{N}, m \in \mathsf{mon}(\Phi)$ *and* $s \in \mathcal{N} \to \Phi(\mathcal{N})$, *then* $\mathsf{CoIt}(m,s,r) \in \nu(\Phi)$.
(7) *If* $\mathcal{N} \in \mathsf{SAT}, r \in \mathcal{N}, m \in \mathsf{mon}(\Phi)$ *and* $s \in \mathcal{N} \to \Phi(\nu(\Phi) + \mathcal{N})$, *then* $\mathsf{CoRec}(m,s,r) \in \nu(\Phi)$.
(8) *If* $m \in \mathsf{mon}(\Phi)$ *and* $r \in \Phi(\nu(\Phi))$, *then* $\mathsf{out}^{-1}(m,r) \in \nu(\Phi)$.

*Proof.* For part (1), we use Lemmas 4.6 and 4.5. For parts (2)–(4) we use Lemmas 4.9 and 4.8. Part (5) follows from Lemmas 4.16 and 4.15. Finally parts (6)–(8) are consequence of Lemmas 4.17 and 4.11. □

### 4.4. Predicates of strong computability

The remainder of the proof of strong normalization follows a standard technique, first we define the so-called strong computability predicates, which are saturated sets that define a semantics for the types. Then we prove substitution and coincidence lemmas for these predicates and prove that every typable term lies in the interpretation of its type as a strong computability predicate and hence belongs to $\mathsf{SN}$. This proves strong normalization.

**Definition 4.17.** A candidate assignment is a finite set of pairs of the form $X : \mathcal{M}$ where $X$ is a type variable and $\mathcal{M} \in \mathsf{SAT}$ such that no type variable occurs twice. Candidate assignments are denoted with $\Delta$. For the candidate $\Delta, X : \mathcal{M}$, it is understood that $X$ does not occur in $\Delta$.

Next, we define for every type $A$ a strong computability predicate depending on a candidate assignment.

**Definition 4.18** (strong computability predicates)**.** Given a type $A$ and a candidate assignment $\Delta$, we define the saturated set of strongly computable terms with

respect to $A$ and $\Delta$, denoted $\mathsf{SC}^A[\Delta]$, as follows:

$$\begin{aligned}
\mathsf{SC}^X[\Delta] &= \begin{cases} \mathcal{M} & \text{if } X : \mathcal{M} \in \Delta \\ \mathsf{SN} & \text{otherwise} \end{cases} \\
\mathsf{SC}^{A\to B}[\Delta] &= \mathsf{SC}^A[\Delta] \to \mathsf{SC}^B[\Delta] \\
\mathsf{SC}^{A+B}[\Delta] &= \mathsf{SC}^A[\Delta] + \mathsf{SC}^B[\Delta] \\
\mathsf{SC}^{A\times B}[\Delta] &= \mathsf{SC}^A[\Delta] \times \mathsf{SC}^B[\Delta] \\
\mathsf{SC}^{\forall X.A}[\Delta] &= \bigcap_{\mathcal{M}\in\mathsf{SAT}} \mathsf{SC}^A[\Delta, X : \mathcal{M}] \\
\mathsf{SC}^{\mu X.A}[\Delta] &= \mu(\Phi_\Delta^{\lambda X.A}) \\
\mathsf{SC}^{\nu X.A}[\Delta] &= \nu(\Phi_\Delta^{\lambda X.A})
\end{aligned}$$

where $\Phi_\Delta^{\lambda X.A} : \mathsf{SAT} \to \mathsf{SAT}$ is defined as:

$$\Phi_\Delta^{\lambda X.A}(\mathcal{M}) = \mathsf{SC}^A[\Delta, X : \mathcal{M}].$$

Observe that for the case of (co)inductive types we do not know if the operator $\Phi_\Delta^{\lambda X.A}$ on saturated sets is monotone. However, due to the developed machinery on saturated sets, originally introduced in [19] for inductive types, and extended here for coinductive types, the monotonicity requirement is dropped. This subtlety makes a big difference with the approach presented in [23] where the definition of saturated sets for fixed-points requires a monotone operator on $\mathsf{SAT}$. The price paid here is the development of complicated constructions on saturated sets whereas the price paid there is the need for a separated proof of monotonicity for the operator $\Phi_\Delta^{\lambda X.A}$, which has to be developed simultaneously with the definition of strong computability predicates.

The following lemmas will allow us to get our desired result.

**Lemma 4.18** (coincidence and substitution)**.** *The following properties hold:*

- *If* $X \notin FV(A)$ *then* $\mathsf{SC}^A[\Delta, X : \mathcal{M}] = \mathsf{SC}^A[\Delta]$.
- $\mathsf{SC}^{A[X:=B]}[\Delta] = \mathsf{SC}^A[\Delta, X : \mathsf{SC}^B[\Delta]]$.

*Proof.* Induction on $A$. □

**Lemma 4.19** (main lemma)**.** *If* $\Gamma \vdash r : A$ *with* $\Gamma = \{x_1 : A_1, \ldots, x_k : A_k\}$ *and for all* $1 \leq i \leq k$ *we have* $s_i \in \mathsf{SC}^{A_i}[\Delta]$*, then* $r[\vec{x} := \vec{s}\,] \in \mathsf{SC}^{A}[\Delta]$.

*Proof.* Induction on $\vdash$. □

We show now that every typable term belongs to the set $\mathsf{SN}$.

**Proposition 4.5.** *If* $\Gamma \vdash r : A$ *then* $r \in \mathsf{SN}$.

*Proof.* Assume $\Gamma = \{x_1 : A_1, \ldots, x_k : A_k\}$. As the set of variables is contained in every saturated set we have $x_i \in \mathsf{SC}^{A_i}[\varnothing]$. Applying the main lemma to the typing $\Gamma \vdash r : A$ we get $r[\vec{x} := \vec{x}] \in \mathsf{SC}^{A}[\varnothing] \subseteq \mathsf{SN}$. Therefore $r \in \mathsf{SN}$. □

**Corollary 4.2.** *Every typable term in* MICT *is strongly normalizing.*

*Proof.* Let $r$ be a term of MICT such that $\Gamma \vdash r : A$ for some context $\Gamma$ and type $A$. We need to prove that $r \in \mathsf{sn}$. By Proposition 4.1 we have $\mathsf{SN} \subseteq \mathsf{sn}$, then it suffices to show that $r \in \mathsf{SN}$, but this is immediate from Proposition 4.5. □

#### 4.4.1. *Type preservation for* MICT

The type-preservation or subject-reduction property is not trivial to prove due to the absence of type annotations on terms characteristic of Curry-style systems. A detailed proof of this property will be given for the second type system presented in this article, proof which can easily be simplified to get a proof for the current system.

## 5. A type system for dialgebras

In this section we develop an extension of F with clausular (co)inductive types modeling the initial and final dialgebras of functor tuples defined in Section 2.3. The system MCICT presented here, can be seen as an extension of Hagino's system $C\lambda$ (see [13]) with parametric polymorphism and using full monotonicity, instead of positivity. The use of clauses avoids sums and products and allows to have multiple constructors/destructors, feature which simplifies programming and modularizes the definition of monotonicity witnesses involved in the typing rules.

### 5.1. Definition of the system

Extend system F as follows:

- Types are generated as

$$A, B, C, F, G \ ::= \ \ldots \mid \mu X(F_1, \ldots, F_k) \mid \nu X(F_1, \ldots, F_k)$$

  Here, every $F_i$ is called a *clause*, and $X$ is bound in both $\mu X(F_1, \ldots, F_k)$ and $\nu X(F_1, \ldots, F_k)$.

- Terms are generated as follows:

$$t, r, s, m \quad ::= \quad \ldots \mid \mathsf{in}_{k,i}\, t \mid \mathsf{It}_k(\vec{m}, \vec{s}, t) \mid \mathsf{Rec}_k(\vec{m}, \vec{s}, t) \mid$$
$$\mathsf{CoIt}_k(\vec{m}, \vec{s}, t) \mid \mathsf{CoRec}_k(\vec{m}, \vec{s}, t) \mid \mathsf{out}_{k,i}\, t \mid \mathsf{out}_k^{-1}(\vec{m}, \vec{t})$$

  where in all cases the length of a vector is $k$.

  The reader may have noticed the absence of a term constructor $\mathsf{in}_k^{-1}$ corresponding to inductive inversion. This omission will be discussed in Section 5.2.
- Typing rules: we add a typing rule for every new term constructor as follows:
  - Introduction of inductive types:

$$\frac{\Gamma \vdash t : F_i[X := \mu X(F_1, \ldots, F_k)]}{\Gamma \vdash \mathsf{in}_{k,i}\, t : \mu X(F_1, \ldots, F_k)} \ (\mu I).$$

  - Elimination of inductive types:
    - By iteration:

$$\frac{\begin{array}{l}\Gamma \vdash t : \mu X(F_1, \ldots, F_k) \\ \Gamma \vdash m_i : F_i \,\mathsf{mon}\, X \ \ 1 \le i \le k \\ \Gamma \vdash s_i : F_i[X := B] \to B \ \ 1 \le i \le k\end{array}}{\Gamma \vdash \mathsf{It}_k(\vec{m}, \vec{s}, t) : B} \ (\mu E).$$

    - By primitive recursion:

$$\frac{\begin{array}{l}\Gamma \vdash t : \mu X(F_1, \ldots, F_k) \\ \Gamma \vdash m_i : F_i \,\mathsf{mon}\, X \ \ 1 \le i \le k \\ \Gamma \vdash s_i : F_i[X := \mu X(F_1, \ldots, F_k) \times B] \to B \ \ 1 \le i \le k\end{array}}{\Gamma \vdash \mathsf{Rec}_k(\vec{m}, \vec{s}, t) : B} \ (\mu E^+).$$

  - Introduction of coinductive types:
    - By coiteration:

$$\frac{\begin{array}{l}\Gamma \vdash s_i : B \to F_i[X := B] \ \ 1 \le i \le k \\ \Gamma \vdash m_i : F_i \,\mathsf{mon}\, X \ \ 1 \le i \le k \\ \Gamma \vdash t : B\end{array}}{\Gamma \vdash \mathsf{CoIt}_k(\vec{m}, \vec{s}, t) : \nu X(F_1, \ldots, F_k)} \ (\nu I).$$

    - By primitive corecursion:

$$\frac{\begin{array}{l}\Gamma \vdash s_i : B \to F_i[X := \nu X(F_1, \ldots, F_k) + B] \ \ 1 \le i \le k \\ \Gamma \vdash m_i : F_i \,\mathsf{mon}\, X \ \ 1 \le i \le k \\ \Gamma \vdash t : B\end{array}}{\Gamma \vdash \mathsf{CoRec}_k(\vec{m}, \vec{s}, t) : \nu X(F_1, \ldots, F_k)} \ (\nu I^+).$$

∗ By coinductive inversion:

$$\frac{\Gamma \vdash t_i : F_i[X := \nu X(F_1,\ldots,F_k)] \;\; 1 \le i \le k \qquad \Gamma \vdash m_i : F_i \,\mathsf{mon}\, X \;\; 1 \le i \le k}{\Gamma \vdash \mathsf{out}_k^{-1}(\vec{m},\vec{t}) : \nu X(F_1,\ldots,F_k)} \;\; (\nu I^i).$$

– Elimination of coinductive types:

$$\frac{\Gamma \vdash r : \nu X(F_1,\ldots,F_k)}{\Gamma \vdash \mathsf{out}_{k,i}\, r : F_i[X := \nu X(F_1,\ldots,F_k)]} \;\; (\nu E).$$

- Operational semantics: it is given by extending the one-step $\beta$-reduction relation $t \to_\beta t'$ with the following axioms under contextual closure, where pairing and copairing of functions is defined exactly as in page 719.

$$\begin{aligned}
\mathsf{It}_k(\vec{m},\vec{s},\mathsf{in}_{k,i}\, t) &\mapsto_\beta s_i\Big(m_i\big(\lambda x.\mathsf{It}_k(\vec{m},\vec{s},x)\big)t\Big)\\
\mathsf{Rec}_k(\vec{m},\vec{s},\mathsf{in}_{k,i}\, t) &\mapsto_\beta s_i\Big(m_i\Big(\langle \mathsf{Id}, \lambda z.\mathsf{Rec}_k(\vec{m},\vec{s},z)\rangle\Big)t\Big)\\
\mathsf{out}_{k,i}\,\mathsf{CoIt}_k(\vec{m},\vec{s},t) &\mapsto_\beta m_i\Big(\lambda z.\mathsf{CoIt}_k(\vec{m},\vec{s},z)\Big)(s_i t)\\
\mathsf{out}_{k,i}\,\mathsf{CoRec}_k(\vec{m},\vec{s},t) &\mapsto_\beta m_i\Big([\mathsf{Id}, \lambda z.\mathsf{CoRec}_k(\vec{m},\vec{s},z)]\Big)(s_i t)\\
\mathsf{out}_{k,i}\,\mathsf{out}^{-1_k}(\vec{m},\vec{t}\,) &\mapsto_\beta m_i(\lambda z.z)t_i.
\end{aligned}$$

Of course, each of the above rules correspond to one of the categorical principles on dialgebras discussed in Section 2.3. In particular observe that the coinductive inversion principle given by equations (2.11) on page 714 is modeled here in a curried way.

This finishes the definition of the system MCICT. A system of *monotone and clausular inductive and coinductive types*.

## 5.2. On the inverse for in

In this section we briefly discuss the absence of a term constructor for inductive inversion $\mathsf{in}_k^{-1}$ in our system MCICT.

Above all, this omission is due to the fact that equation (2.14) on page 715 is not suitable to be represented directly in our framework. Observe that the inverse of the $\mathsf{in}_k$ morphism given in page 715 is a morphism $\mathsf{in}_k^{-1} : \langle \mu,\ldots,\mu\rangle \to \langle F_1\mu,\ldots,F_k\mu\rangle$ such that $\mathsf{in}_k^{-1} \circ \langle \mathsf{in}_{k,1},\ldots,\mathsf{in}_{k,k}\rangle = \mathsf{Id}_{\langle F_1\mu,\ldots,F_n\mu\rangle}$.

Therefore we would need a typing rule like the following:

$$\frac{\Gamma \vdash t : \big\langle \mu X(F_1,\ldots,F_k),\ldots,\mu X(F_1,\ldots,F_k)\big\rangle \quad \Gamma \vdash m_i : F_i \,\mathsf{mon}\, X, \;\; 1 \le i \le k}{\Gamma \vdash \mathsf{in}_k^{-1}(\vec{m},t) : \Big\langle F_1[X := \mu X(F_1,\ldots,F_k)],\ldots,F_k[X := \mu X(F_1,\ldots,F_k)]\Big\rangle}.$$

To admit this kind of rule we would need to handle a tuple of types as a single type and, although this issue can be fulfilled using variant types, it would complicate the system only to be able to model this principle.

On the other hand the main application of such rule – to define inductive destructors – can be done by means of the following reasoning obtained by inversion of an instance of the typing rule ($\mu I$): "If we have an inductive object $t : \mu X(F_1, \ldots, F_k)$ then it was generated by a clause $\mathsf{in}_k^{-1}\, t : F_i[X := \mu X(F_1, \ldots, F_k)]$ for some $1 \leq i \leq k$". This rule is used for instance, to guarantee that if $t$ is a natural number then $t$ is either 0 or a successor $\mathsf{suc}\, n$. Such reasoning corresponds to an inverse $\mathsf{in}_k^{-1} : \mu \to F_1\mu + \ldots + F_k\mu$ such that $\mathsf{in}_k^{-1}(\vec{m}, \mathsf{in}_{k,i}\, t) \;=\; \mathsf{inj}_i^k \left(m_i(\lambda z.z)t\right)$, where $\mathsf{inj}_i^k$ is the canonical $i$th-injection and can be modeled by the rule:

$$\frac{\Gamma \vdash t : \mu X(F_1, \ldots, F_k) \quad \Gamma \vdash m_i : F_i \,\mathsf{mon}\, X, \;\; 1 \leq i \leq k}{\Gamma \vdash \mathsf{in}_k^{-1}(\vec{m}, t) : F_1[X := \mu X(F_1, \ldots, F_k)] + \ldots + F_k[X := \mu X(F_1, \ldots, F_k)]}(\mu E^i).$$

But we observe that the task of defining destructors on inductive types, which is the main reason to add the rule to the system, can easily be achieved using primitive recursion, a principle already present in the system. This claim should be made clear with the examples below. Therefore we will omit the rule as it would cause more problems than profits; its main disadvantage being the generation of a term inhabiting a sum type in an unusual way.

### 5.3. Programming in MCICT

In this section we develop several examples of programming in MCICT. The presentation is similar to the one in Section 3.2. Let us briefly explain the methodology of function definition.

In the case of a function $g : \mu X(F_1, \ldots, F_k) \to A$, the iteration principle ensures the existence of a program for $g$ if $g$ is defined by the following recurrence equations:

$$\begin{array}{rcl} g(\mathsf{in}_{k,1}\, x) & = & s_1\, (m_1\, g\, x) \\ & \vdots & \\ g(\mathsf{in}_{k,k}\, x) & = & s_k\, (m_k\, g\, x) \end{array}$$

where $s_i : F_i[X := A] \to A$ and $m_i : F_i \,\mathsf{mon}\, X, 1 \leq i \leq k$ are the fixed monotonicity witnesses used to eliminate the type $\mu X(F_1, \ldots, F_k)$. If these conditions hold, then the categorical machinery says that we can define $g = \lambda z.\mathsf{It}_k(\vec{m}, \vec{s}, z)$ and we will obtain the desired reduction behavior:

$$g(\mathsf{in}_{k,i}\, x) \to_\beta^+ s_i\, (m_i\, g\, x).$$

Analogously primitive recursion provides a means to program functions $g : \mu X(F_1, \ldots, F_k) \to A$ which satisfy the following recurrence equations:

$$\begin{array}{rcl} g(\mathsf{in}_{k,1}\, x) & = & s_1\ (m_1\ \langle \mathsf{Id}, g\rangle\ x) \\ & \vdots & \\ g(\mathsf{in}_{k,k}\, x) & = & s_k\ ((m_k\ \langle \mathsf{Id}, g\rangle\ x) \end{array}$$

with $s_i : F_i[X := \mu X(F_1, \ldots, F_k) \times A] \to A$. In this case, $g$ can be defined as $g = \lambda z.\mathsf{Rec}_k(\vec{m}, \vec{s}, z)$.

In a dual way we can program a function $g : A \to \nu X(F_1, \ldots, F_k)$ which satisfies the following coiteration equations:

$$\begin{array}{rcl} \mathsf{out}_{k,1}(gx) & = & (m_1\ g)\ (s_1\ x) \\ & \vdots & \\ \mathsf{out}_{k,k}(gx) & = & (m_k\ g)\ (s_k\ x) \end{array}$$

where $s_i : A \to F_i[X := A]$ and $m_i : F_i\ \mathsf{mon}\ X, 1 \leq i \leq k$ are the fixed monotonicity witnesses used to introduce the type $\nu X(F_1, \ldots, F_k)$. In this case, the program is $g = \lambda z.\mathsf{CoIt}_k(\vec{m}, \vec{s}, z)$.

Finally, the corecursion principle provides a means to program functions $g : A \to \nu X(F_1, \ldots, F_k)$ which satisfies the following equations:

$$\begin{array}{rcl} \mathsf{out}_{k,1}(g\ x) & = & (m_1\ [\mathsf{Id}, g])\ (s_1\ x) \\ & \vdots & \\ \mathsf{out}_{k,k}(g\ x) & = & (m_k\ [\mathsf{Id}, g])\ (s_k\ x) \end{array}$$

with $s_i : A \to F_i[X := \nu X(F_1, \ldots, F_k) + A]$. In this case, a program is $g = \lambda z.\mathsf{CoRec}_k(\vec{m}, \vec{s}, z)$.

Next we show several examples of programming in MCICT. The reader is invited to program more functions and to verify the soundness of programs with respect to the operational semantics.

Let us start with the degenerated (co)inductive types having no clauses.

**Example 5.1** (degenerated (co)inductive types)**.** The inductive type with no clauses represents the empty type $\mathsf{0} = \mu X()$. As there are no clauses available, the iteration rule becomes

$$\frac{\Gamma \vdash t : \mathsf{0}}{\Gamma \vdash \mathsf{It}_0(t) : B}\ (0E)$$

and therefore $\mathsf{0}$ cannot be inhabited.

On the other hand, for the coinductive type with no clauses $\nu X()$ the rules for coiteration and corecursion degenerate as follows:

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash \mathsf{CoIt}_0(t) : \nu X()} \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \mathsf{CoRec}_0(t) : \nu X()}$$

for every type $B$. Observing that type $B$ does not play an important role, we can say that $\nu X()$ has essentially two inhabitants, and convey to define $2 = \nu X()$ with inhabitants $\mathsf{CoIt}_0(\star), \mathsf{CoRec}_0(\star)$.

The next three examples correspond to the naturals, finite lists and streams already implemented in Examples 3.3, 3.4 and 3.7. We repeat them here to allow a direct comparison on the programming facilities of both systems.

**Example 5.2** (the natural numbers)**.** We define $\mathsf{Nat} = \mu X(1, X)$.

- Canonical monotonicity witnesses: $\mathsf{mapn}_1 = \lambda f \lambda x.x$, $\mathsf{mapn}_2 = \lambda x.x$.
- Constructors:
  - Zero: $0 : \mathsf{Nat}$, $0 = \mathsf{in}_{2,1} \star$.
  - Successor function: $\mathsf{suc} : \mathsf{Nat} \to \mathsf{Nat}$, $\mathsf{suc} = \mathsf{in}_{2,2}$.
- Destructors: as we do not have inductive inversion in the system, we define the destructor by recursion, $\mathsf{pred} : \mathsf{Nat} \to 1 + \mathsf{Nat}$ such that $\mathsf{pred}\ 0 = \mathsf{error}$, $\mathsf{pred}\,(\mathsf{suc}\, n) = \mathsf{inr}\, n$, given by $\mathsf{pred} = \lambda n.\mathsf{Rec}_2(\mathsf{mapn}_1, \mathsf{mapn}_2, \lambda\_.0, \lambda z.\, \mathsf{fst}\, z, n)$.
- Some functions on $\mathsf{Nat}$:
  - $\mathsf{sum} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$, $\mathsf{sum} = \lambda n \lambda m.\mathsf{It}_2(\mathsf{map}_1, \mathsf{map}_2, \lambda\_.n, \mathsf{suc}, m)$.
  - $\mathsf{prod} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$, $\mathsf{prod} = \lambda n \lambda m.\mathsf{It}_2(\mathsf{map}_1, \mathsf{map}_2, \mathsf{in}_{2,1}, \lambda y.\mathsf{sum}\, y\, n, m)$.

**Example 5.3** (finite lists over $A$)**.** This type is defined as $\mathsf{List}\, A = \mu X(1, A \times X)$.

- Canonical monotonicity witnesses:
  $\mathsf{mapl}_1 = \lambda f \lambda x.x$, $\mathsf{mapl}_2 = \lambda f \lambda x.\langle \mathsf{fst}\, x, f(\mathsf{snd}\, x)\rangle$.
- Constructors:
  - Empty list: $\mathsf{nil} : \mathsf{List}\, A$, $\mathsf{nil} = \mathsf{in}_{2,1} \star$.
  - Cons function: $\mathsf{cons} : A \times \mathsf{List}\, A \to \mathsf{List}\, A$, $\mathsf{cons} = \mathsf{in}_{2,2}$.
- Destructors:
  - Head function: $\mathsf{head} : \mathsf{List}\, A \to 1 + A$ such that $\mathsf{head}\, \mathsf{nil} = \mathsf{error}$, $\mathsf{head}(\mathsf{cons}\langle a, \ell\rangle) = \mathsf{inr}\, a$, given by $\mathsf{head} = \lambda z.\mathsf{Rec}(\mathsf{mapl}_1, \mathsf{mapl}_2, \lambda x.\, \mathsf{inl}\, x, \lambda y.\, \mathsf{inr}(\mathsf{fst}\, y), z)$.
  - Tail function: $\mathsf{tail} : \mathsf{List}\, A \to 1 + \mathsf{List}\, A$ such that $\mathsf{tail}\, \mathsf{nil} = \mathsf{error}$, $\mathsf{tail}(\mathsf{cons}\langle a, \ell\rangle) = \mathsf{inr}\, \ell$, given by $\mathsf{tail} = \lambda z.\mathsf{Rec}(\mathsf{mapl}_1, \mathsf{mapl}_2, \lambda x.\, \mathsf{inl}\, x, \lambda y.\, \mathsf{inr}(\mathsf{fst}(\mathsf{snd}\, y))), z)$.
- Some functions on $\mathsf{List}\, A$:
  - Append: $\mathsf{app} : \mathsf{List}\, A \to \mathsf{List}\, A \to \mathsf{List}\, A$
    $\mathsf{app} = \lambda x.\mathsf{It}_2(\mathsf{mapl}_1, \mathsf{mapl}_2, \lambda y \lambda z z, \lambda u \lambda v.\, \mathsf{cons}\langle \mathsf{fst}\, u, (\mathsf{snd}\, u) v\rangle)$.
  - Length: $\mathsf{length} : \mathsf{List}\, A \to \mathsf{Nat}$
    $\mathsf{length} = \lambda x.\mathsf{It}_2(\mathsf{mapl}_1, \mathsf{mapl}_2, \lambda x.0, \lambda y.\, \mathsf{suc}(\mathsf{snd}\, y), x)$.

– Reverse: $\mathsf{rev} : \mathsf{List}\, A \to \mathsf{List}\, A$
$\mathsf{rev} = \lambda x.\mathsf{It}_2(\mathsf{mapl}_1, \mathsf{mapl}_2, \mathsf{nil}, \lambda z.\mathsf{app}\,(\mathsf{snd}\, z)\,(\mathsf{cons}\langle \mathsf{fst}\, z, \mathsf{nil}\rangle), x)$.
– The polymorphic map function on lists: $\mathsf{maplist} : \forall X \forall Y.(X \to Y) \to \mathsf{List}\, X \to \mathsf{List}\, Y$ such that:

$$\mathsf{maplist} f\ \mathsf{nil} = \mathsf{nil} \qquad \mathsf{maplist} f\ (\mathsf{cons}\langle x, xs\rangle) = \mathsf{cons}\,\langle f\, x, \mathsf{maplist} f\, xs\rangle.$$

The program is:

$$\mathsf{maplist} = \lambda f \lambda x.\mathsf{It}_2(\mathsf{mapl}_1, \mathsf{mapl}_2, \lambda u.\mathsf{nil}, \lambda v.\, \mathsf{cons}\langle f\, (\mathsf{fst}\, v), \mathsf{snd}\, v\rangle, x).$$

Next, we define some interesting coinductive types.

**Example 5.4** (streams (infinite lists) over $A$)**.** We define $\mathsf{Stream}\, A = \nu X(A, X)$

- Canonical monotonicity witnesses: $\mathsf{maps}_1 = \lambda f \lambda x.x,\ \mathsf{maps}_2 = \lambda f.f$.
- Destructors:
  – $\mathsf{head} : \mathsf{Stream}\, A \to A,\ \ \mathsf{head} = \mathsf{out}_{2,1}$
  – $\mathsf{tail} : \mathsf{Stream}\, A \to \mathsf{Stream}\, A,\ \ \mathsf{tail} = \mathsf{out}_{2,2}$.
- Constructor $\mathsf{cons} : A \to \mathsf{Stream}\, A \to \mathsf{Stream}\, A$. We have two choices
  – By corecursion: $\mathsf{cons} = \lambda x \lambda \ell.\mathsf{CoRec}_2(\mathsf{maps}_1, \mathsf{maps}_2, \lambda\_.x, \lambda z.\, \mathsf{inl}\, z, \ell)$.
  – By coinductive inversion: $\mathsf{cons} = \lambda x \lambda \ell.\, \mathsf{out}_2^{-1}(\mathsf{maps}_1, \mathsf{maps}_2, x, \ell)$.
- Some functions:
  – Streams of constants, $\mathsf{cnt} : A \to \mathsf{Stream}\, A$ such that $\mathsf{head}(\mathsf{cnt}\, a) = a,\ \mathsf{tail}(\mathsf{cnt} a) = \mathsf{cnt}\, a$
  $\mathsf{cnt} = \lambda x.\mathsf{CoIt}_2(\mathsf{maps}_1, \mathsf{maps}_2, \lambda z z, \lambda z z, x)$.
  – The stream of natural numbers from a given one:
  $\mathsf{from} : \mathsf{Nat} \to \mathsf{Stream}\, \mathsf{Nat},\ \ \mathsf{head}(\mathsf{from}\, n) = n,\ \mathsf{tail}(\mathsf{from}\, n) = \mathsf{from}(\mathsf{suc}\, n)$
  $\mathsf{from} = \lambda x.\mathsf{CoIt}_2(\mathsf{maps}_1, \mathsf{maps}_2, \lambda z z, \mathsf{suc}, x)$.
  – A zip function for streams, $\mathsf{zip} : \mathsf{Stream}\, A \times \mathsf{Stream}\, B \to \mathsf{Stream}(A \times B)$ specified by: $\mathsf{head}(\mathsf{zip}\langle \ell_1\, \ell_2\rangle) = \langle \mathsf{head}\, \ell_1, \mathsf{head}\, \ell_2\rangle,\ \ \mathsf{tail}(\mathsf{zip}\langle \ell_1\, \ell_2\rangle) = \mathsf{zip}\,\langle \mathsf{tail}\, \ell_1\ \mathsf{tail}\, \ell_2\rangle$. A coiterative implementation is $\mathsf{zip} = \lambda z.\mathsf{CoIt}_2(\mathsf{maps}_1, \mathsf{maps}_2, s_1, s_2, z)$ where $s_1 = \lambda x.\langle \mathsf{head}(\mathsf{fst}\, x), \mathsf{head}(\mathsf{snd}\, x)\rangle, s_2 = \lambda x.\langle \mathsf{tail}\, (\mathsf{fst}\, x), \mathsf{tail}(\mathsf{snd}\, x)\rangle$.
  – The polymorphic maphead function $\mathsf{maphead} : \forall X.(X \to X) \to \mathsf{Stream}\ X \to \mathsf{Stream}\, X$ such that $\mathsf{head}(\mathsf{maphead}\, f\, \ell) = f(\mathsf{head}\, \ell)$, $\mathsf{tail}(\mathsf{maphead}\, f\, \ell) = \mathsf{tail}\, \ell$ can easily be implemented by corecursion as $\mathsf{maphead} = \lambda f \lambda \ell.\mathsf{CoRec}_2(\mathsf{maps}_1, \mathsf{maps}_2, f \circ \mathsf{head}, \mathsf{inr} \circ \mathsf{tail}, \ell)$.

The next example is an implementation of Example 2.4.

**Example 5.5.** The codatatype of non-empty, and maybe, infinite lists of elements of $C$ is defined as $\mathsf{Colist}\, C = \nu X(C, 1 + X)$.

- Canonical monotonicity witnesses:
  $\mathsf{mapcl}_1 = \lambda f.\lambda x.x,\ \mathsf{mapcl}_2 = \lambda f \lambda x.\mathsf{case}(x, y.\, \mathsf{inl}\, y, z.\, \mathsf{inr}(f\, z))$
- Destructors:
  – $\mathsf{head} : \mathsf{Colist}\, C \to C,\ \ \mathsf{head} = \mathsf{out}_{2,1}$. Observe that $\mathsf{head}\ \ell$ is always an element of $C$, for the empty list is not present.

  - $\mathsf{tail} : \mathsf{Colist}\, C \to 1 + \mathsf{Colist}\, C,\ \ \mathsf{tail} = \mathsf{out}_{2,2}$. If a colist is finite, then the repeated application of $\mathsf{tail}$ to it will eventually return an error indicating that we have reached its end.
- Constructors: we have a general constructor $\mathsf{cons}$ and a constructor of singleton lists $\mathsf{single}$.
  - $\mathsf{cons} : C \to \mathsf{Colist}\, C \to \mathsf{Colist}\, C,\ \mathsf{cons} = \lambda x \lambda \ell.\, \mathsf{out}_2^{-1}(\mathsf{mapcl}_1, \mathsf{mapcl}_2, x, \mathsf{inr}\, \ell)$
  - $\mathsf{single} : C \to \mathsf{Colist}\, C,\ \mathsf{single} = \lambda x.\, \mathsf{out}_2^{-1}(\mathsf{mapcl}_1, \mathsf{mapcl}_2, x, \mathsf{inl}\, \star)$.
- Some functions:
  - A test for single lists $\mathsf{issingle} : \mathsf{Colist}\, C \to \mathsf{Bool}$ is given by $\mathsf{issingle} = \lambda \ell.\mathsf{case}(\mathsf{tail}\, \ell, y.\mathsf{true}, z.\mathsf{false})$. This definition yields $\mathsf{issingle}\, \ell = \mathsf{true}$ if and only if $\ell = \mathsf{single}\, x$ for some $x : C$.
  - Colists of numbers in a given interval of naturals: $\mathsf{from} : \mathsf{Nat} \to \mathsf{Nat} \to 1 + \mathsf{Colist}\, \mathsf{Nat}$. We first define a function $\mathsf{sfrom} : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Colist}\, \mathsf{Nat}$ such that for $n < m$, $\mathsf{head}(\mathsf{sfrom}\, \langle n, m \rangle) = n$ and $\mathsf{tail}(\mathsf{sfrom}\, \langle n, m \rangle) = \mathsf{inr}(\mathsf{sfrom}\, \langle n{+}1, m \rangle)$, by means of $\mathsf{sfrom} = \lambda z.\mathsf{CoIt}_2\, (\mathsf{mapcl}_1, \mathsf{mapcl}_2, s_1, s_2, z)$, where $s_1 = \lambda \_.n,\ s_2 = \lambda w.\, \mathsf{inr}\langle (\mathsf{fst}\, w) + 1, \mathsf{snd}\, w \rangle$. The desired function $\mathsf{from}$ is then defined as: $\mathsf{from}\, n\, n = \mathsf{inr}(\mathsf{single}\, n)$ and otherwise $\mathsf{from}\ n\, m = \mathsf{if}\ n < m\ \mathsf{then}\ \mathsf{inr}(\mathsf{sfrom}\langle n,\ m \rangle)\ \mathsf{else}\ \mathsf{error}$
  - The polymorphic map function on colists: $\mathsf{mapcls} : \forall X \forall Y.(X \to Y) \to \mathsf{Colist}\, X \to \mathsf{Colist}\, Y$ is specified by $\mathsf{head}(\mathsf{mapcls}\, f\, \ell) = f\, (\mathsf{head}\, \ell)$, $\mathsf{tail}(\mathsf{mapcls}\, f\, \ell) = \mathsf{if}\ \mathsf{issingle}\, \ell\ \mathsf{then}\ \mathsf{error}\ \mathsf{else}\ \mathsf{inr}(\mathsf{mapcls}\, f\, tl)$, where $\mathsf{tail}\, \ell = \mathsf{inr}\ tl$.
    A coiterative implementation is: $\mathsf{mapcls}\, f = \lambda \ell.\mathsf{CoIt}(\mathsf{maps}_1, \mathsf{maps}_2, \lambda x.f\, (\mathsf{head}\, x),\ s_2,\ \ell)$, where $s_2\, \ell = \mathsf{if}\ \mathsf{issingle}\, \ell\ \mathsf{then}\ \mathsf{error}\ \mathsf{else}\ \mathsf{tail}\, \ell$.

Let us discuss now a couple of examples with more than two clauses in their definition.

**Example 5.6** (strictly infinite complete $A$-labelled binary trees)**.** We can define this type as $\nu X(A, X \times X)$ entailing two destructors, one for the label of the root and the other for the two children of a tree. However we can do better by using three clauses to define $\mathsf{Infbtree}\, A = \nu X(A, X, X)$.

- Canonical monotonicity witnesses:
  $\mathsf{mapibt}_1 = \lambda f \lambda x.x,\ \mathsf{mapibt}_2 = \mathsf{mapibt}_3 = \lambda f.f$.
- Destructors:
  - $\mathsf{rlabel} : \mathsf{Infbtree}\, A \to A,\ \ \mathsf{label} = \mathsf{out}_{3,1}$
  - $\mathsf{lsbt} : \mathsf{Infbtree}\, A \to \mathsf{Infbtree}\, A,\ \mathsf{lsbt} = \mathsf{out}_{3,2}$
  - $\mathsf{rsbt} : \mathsf{Infbtree}\, A \to \mathsf{Infbtree}\, A,\ \mathsf{rsbt} = \mathsf{out}_{3,3}$.
- Constructor: $\mathsf{mkibt} : A \to \mathsf{Infbtree}\, A \to \mathsf{Infbtree}\, A \to \mathsf{Infbtree}\, A$ defined by coinductive inversion as $\mathsf{mkibt} = \lambda x \lambda y \lambda z.\, \mathsf{out}_3^{-1}(\mathsf{mapibt}_1, \mathsf{mapibt}_2, \mathsf{mapibt}_3, x, y, z)$.
- A function $\mathsf{swapt} : \mathsf{Infbtree}\, A \to \mathsf{Infbtree}\, A$ that swaps the left and right subtrees of a given tree is specified as follows: $\mathsf{rlabel}\, (\mathsf{swapt}\, t) = \mathsf{rlabel}\, t$, $\mathsf{lsbt}\, (\mathsf{swapt}\, t) = \mathsf{rsbt}\, t,\ \mathsf{rsbt}\, (\mathsf{swapt}\, t) = \mathsf{lsbt}\, t$. This definition cannot be

directly implemented by coiteration. Nevertheless, a straightforward implementation is obtained by corecursion:
$\mathsf{swapt} = \lambda z.\mathsf{CoRec}_3(\mathsf{mapibt}_1, \mathsf{mapibt}_2, \mathsf{mapibt}_3, \mathsf{rlabel}, \mathsf{inl} \circ \mathsf{rsbt}, \mathsf{inl} \circ \mathsf{rsbt}, z)$.

**Example 5.7** (finite and infinite $A$-labelled binary trees)**.** A type for maybe infinite binary trees is defined as $\mathsf{FinInfBTree}\, A = \nu X(A, 1 + X, 1 + X)$.

- Canonical monotonicity witnesses: $\mathsf{mapfibt}_1 = \lambda f \lambda x.x$,
  $\mathsf{mapfibt}_2 = \mathsf{mapfibt}_3 = \lambda f \lambda x.\mathsf{case}(x,\ y.\, \mathsf{inr}\, y,\ z.f(\mathsf{inr}\, z))$.
- Destructors:
  - $\mathsf{rlabel} : \mathsf{FinInfBTree}\, A \to A,\ \ \mathsf{label} = \mathsf{out}_{3,1}$
  - $\mathsf{lsbt} : \mathsf{FinInfBTree}\, A \to 1 + \mathsf{FinInfBTree}\, A,\ \mathsf{lsbt} = \mathsf{out}_{3,2}$
  - $\mathsf{rsbt} : \mathsf{FinInfBTree}\, A \to 1 + \mathsf{FinInfBTree}\, A,\ \mathsf{rsbt} = \mathsf{out}_{3,3}$.

  The destructors for subtrees are able to return an error indicating that the subtree does not exist. This allows to build non complete trees.
- Constructor: $\mathsf{mkbt} : A \to 1 + \mathsf{FinInfBTree}\, A \to 1 + \mathsf{FinInfBTree}\, A \to \mathsf{FinInfBTree}\, A$ defined by coinductive inversion as
  $$\mathsf{mkbt} = \lambda x \lambda y \lambda z.\, \mathsf{out}_3^{-1}(\mathsf{mapibt}_1, \mathsf{mapibt}_2, \mathsf{mapibt}_3, x, y, z).$$
  Observe that if we pass $\mathsf{inl}\,\star$ as the second or third argument to $\mathsf{mkbt}$ we will build a tree with no left or right subtree.
- A polymorphic function $\mathsf{copy} : \forall X.X \to \mathsf{FinInfBTree}\, X \to \mathsf{FinInfBTree}\, X$, which for a given label $a$ and tree $t$ builds a new tree with root label $a$ and $t$ as its subtrees, is specified by $\mathsf{rlabel}\,(\mathsf{copy}\, a\, t) = a,\ \mathsf{lsbt}\,(\mathsf{copy}\, a\, t) = \mathsf{inr}\, t,\ \mathsf{rsbt}\,(\mathsf{copy}\, a\, t) = \mathsf{inr}\, t$. An implementation by corecursion is
  $\mathsf{copy} = \lambda x.\lambda y.\mathsf{CoRec}_3(\mathsf{mapfibt}_1, \mathsf{mapfibt}_2, \mathsf{mapfibt}_3, \lambda\_.x, \mathsf{inr} \circ \mathsf{inl},\ \mathsf{inr} \circ \mathsf{inl},\ y)$.

**Example 5.8** (finite branching, $A$-labelled trees with potentially infinite depth)**.** $\mathsf{Pidtree}\, A = \nu X(A, \mathsf{List}\, X)$ with destructors

- $\mathsf{rlabel} : \mathsf{Pidtree}\, A \to A,\ \mathsf{rlabel} = \mathsf{out}_{2,1}$
- $\mathsf{lstrees} : \mathsf{Pidtree}\, A \to \mathsf{List}(\mathsf{Pidtree}\, A),\ \mathsf{lstrees} = \mathsf{out}_{2,2}$.

An inhabitant $t : \mathsf{Pidtree}\, A$ is destructed as the label of its root $\mathsf{rlabel}\, t$ and the list of its immediate subtrees $\mathsf{lstrees}\, t$.

- Canonical monotonicity witnesses: $\mathsf{mappid}_1 = \lambda f \lambda x.x,\ \mathsf{mappid}_2 = \mathsf{maplist}$.
  This example shows the important connection between the inductive and the coinductive part of our system. The coinductive subsystem depends on the inductive one to be able to define the monotonicity witness of a coinductive type. In this case the iteratively defined witness $\mathsf{maplist}$ for the coinductive type $\mathsf{Pidtree}\, A$.
- The function $\mathsf{maptree} : (A \to C) \to \mathsf{Pidtree}\, A \to \mathsf{Pidtree}\, C$, mapping a function $f : A \to C$ into a tree $t : \mathsf{Pidtree}\, A$ is coiteratively defined by:
  $$\begin{aligned} \mathsf{rlabel}\,(\mathsf{maptree}\ f\, t) &= f\,(\mathsf{rlabel}\, t) \\ \mathsf{lstrees}\,(\mathsf{maptree}\ f\, t) &= \mathsf{maplist}\,(\mathsf{maptree}\ f)\,(\mathsf{lstrees}\, t). \end{aligned}$$
  A program for $\mathsf{maptree}\, f$ is:
  $$\mathsf{maptree}\ f = \lambda x.\mathsf{CoIt}_2(\mathsf{mappid}_1,\ \mathsf{mappid}_2,\ \lambda y.f\,(\mathsf{rlabel}\, y),\ \mathsf{lstrees},\ x).$$

The next example employs a function type in a clause and provides the fundamentals of the coalgebraic automata theory developed in [32].

**Example 5.9** (deterministic automata with input alphabet $\Sigma$ and output type $B$)**.**

$$\mathsf{daut}(\Sigma, B) = \nu X(\Sigma \to X, B).$$

- Canonical monotonicity witnesses: $\mathsf{mapda}_1 = \lambda f \lambda g \lambda x.f(gx), \mathsf{mapda}_2 = \lambda f \lambda x.x$.
- Destructors:
  - $\mathsf{next} : \mathsf{daut}(\Sigma, B) \to (\Sigma \to \mathsf{daut}(\Sigma, B)),\ \mathsf{next} = \mathsf{out}_{2,1}$
  - $\mathsf{obs} : \mathsf{daut}(\Sigma, B) \to B,\ \mathsf{obs} = \mathsf{out}_{2,2}$.

In this setting an automata is a pair $M = \langle \delta, o \rangle$ where $\delta : Q \to \Sigma \to Q$ is the transition function, with state space $Q$, and $o : Q \to B$ is an observation function, thus $M$ is a Moore automata. There is no need, neither for the existence of an initial state, nor for assuming that $Q$ or $\Sigma$ are finite.

The coinductive type $\mathsf{daut}(\Sigma, B)$ is inhabited essentially by behavior functions $\mathsf{beh}\, q : \Sigma^\star \to B$ for a given state $q \in Q$.

We can codify such an automata by means of a function $\mathsf{caut} : Q \to \mathsf{daut}(\Sigma, B)$ defined coiteratively:

$$\mathsf{caut} = \lambda z.\mathsf{Colt}_2(\mathsf{mapda}_1, \mathsf{mapda}_2, \delta, o, z)$$

which is destructed as follows:

$$\mathsf{next}(\mathsf{caut}\, q) = \lambda x.\mathsf{caut}((\delta q)x) \qquad \mathsf{obs}(\mathsf{caut}\, q) = oq.$$

Given a behavior function $\mathsf{beh}\, q : \Sigma^\star \to B$ we can codify it by an inhabitant of $\mathsf{daut}(\Sigma, B)$ by means of a function $\mathsf{cbeh} : (\Sigma^\star \to B) \to \mathsf{daut}(\Sigma, B)$ given by:

$$\mathsf{cbeh} = \lambda z.\mathsf{Colt}_2(\mathsf{mapda}_1, \mathsf{mapda}_2, \lambda f \lambda a \lambda w.f(a.w), \lambda g.g(\varepsilon), z)$$

where $\varepsilon, a.w$ denote the nil and cons operations respectively on $\Sigma^\star$. This function is destructed as follows:

$$\mathsf{obs}(\mathsf{cbeh}\,(\mathsf{beh}\, q)) = \mathsf{beh}\, q\, \varepsilon \qquad \mathsf{next}(\mathsf{cbeh}\,(\mathsf{beh}\, q)) = \lambda a.\mathsf{cbeh}\big(\lambda w.\mathsf{beh}\, q\,(a.w)\big).$$

We can define in a similar way other types for automata:

- Partial automata: $\mathsf{paut}(\Sigma, B) = \nu X(\Sigma \to 1 + X, B)$.
  In this case the transition functions are of the form $\delta : Q \to \Sigma \to 1 + Q$ such that $\delta\, q\, a = \mathsf{error}$ if and only if such transition is undefined.
- Finite deterministic automata: just take $Q, \Sigma$ to be finite and $B = \mathsf{Bool}$.

  $$\mathsf{fda}(\Sigma) = \mathsf{daut}(\Sigma, \mathsf{Bool}) = \nu X(\Sigma \to X, \mathsf{Bool}).$$

  Some functions on this type are:

– The complement of an FDA: $\mathsf{comp} : \mathsf{fda}(\Sigma) \to \mathsf{fda}(\Sigma)$ destructed as:

$$\mathsf{next}\,(\mathsf{comp}\,M) = \mathsf{next}\,M \qquad \mathsf{obs}\,(\mathsf{comp}\,M) = \mathsf{not}\,(\mathsf{obs}\,M).$$

– The product automata of two FDA's: $\mathsf{prod} : \mathsf{fda}(\Sigma) \to \mathsf{fda}(\Sigma) \to \mathsf{fda}(\Sigma)$ destructed as follows:

$$\mathsf{next}\,(\mathsf{prod}\,M_1\,M_2) = \mathsf{prod}\,(\mathsf{next}\;M_1)\,(\mathsf{next}\;M_2).$$

According to the language we want to recognize we have different possibilities for the observation function:

$$\mathsf{obs}\,(\mathsf{prod}\,M_1\,M_2) = (\mathsf{obs}\,M_1)\,\mathsf{and}\,(\mathsf{obs}\,M_2), \quad \text{for} \;\; L(M_1) \cap L(M_2)$$

$$\mathsf{obs}\,(\mathsf{prod}\,M_1\,M_2) = (\mathsf{obs}\,M_1)\,\mathsf{or}\,(\mathsf{obs}\,M_2), \quad \text{for} \;\; L(M_1) \cup L(M_2)$$

$$\mathsf{obs}\,(\mathsf{prod}\,M_1\,M_2) = (\mathsf{obs}\,M_1)\,\mathsf{and}\,\mathsf{not}\,(\mathsf{obs}\,M_2), \quad \text{for} \;\; L(M_1) - L(M_2).$$

Our final example is adapted from a similar one given in [15].

**Example 5.10** (potentially infinite trees with $A$-labelled branches)**.** The type $\mathsf{BLTree}\,A = \nu X(\mathsf{List}(A \times X))$ entails trees $t : \mathsf{BLTree}\,A$ of possible infinite depth, with finitely many ordered branches, each provided with a label from a constant type $A$. The destructor $\mathsf{lsb} : \mathsf{BLTree}\,A \to \mathsf{List}(A \times \mathsf{BLTree}\,A)$ given by $\mathsf{lsb} = \mathsf{out}_{1,1}$, returns the list of branches pendant from the root of a tree $t$, *i.e.*, it returns a list of pairs of $A \times \mathsf{BLTree}\,A$ consisting of a label $a$ for the branch and the list of subtrees pendant from that branch in order from left to right.

- Canonical monotonicity witness: the canonical witness

$$\mathsf{mapbltree} : \forall X \forall Y.(X \to Y) \to \mathsf{List}(A \times X) \to \mathsf{List}(A \times Y)$$

is defined as follows:

$$\mathsf{mapbltree}\,f\,\mathsf{nil} = \mathsf{nil} \qquad \mathsf{mapbltree}\,f\,(\mathsf{cons}\langle\langle a, x\rangle, xs\rangle) = \mathsf{cons}\langle\langle a, fx\rangle, \mathsf{mapbltree}\,f\,xs\rangle.$$

But this is easily defined as the application of the iteratively defined $\mathsf{maplist}$ function of Example 5.3 to the function $G\,f$, where $G : (A \times X)\,\mathsf{mon}\,X$ is such that for $f : X \to Y$ we get $G\,f\,\langle a, x\rangle = \langle a, f\,x\rangle$. Again, an inductive definition is required to define a monotonicity witness for a coinductive type.

- Breadth first search: we will program a function $\mathsf{bfs} : \mathsf{BLTree}\,A \to A^\infty$, where $A^\infty = \nu X(1 + A, 1 + X)$ is the type of finite and infinite lists over $A$,

which takes a tree $t$ and returns a list of the labels of branches, in breadth-first order. To do this, we first define a function $\mathsf{bfl} : \mathsf{List}(A \times \mathsf{BLTree}\, A) \to A^{\infty}$ and set $\mathsf{bfs} = \mathsf{bfl} \circ \mathsf{lsb}$. The coinductive specification of $\mathsf{bfl}$ is:

$$\mathsf{head}(\mathsf{bfl}\, t) = \mathsf{if}\ (\mathsf{isnil}\, (\mathsf{bfl}\, t))\ \mathsf{then\ error\ else}\ \mathsf{inr}(\mathsf{fst}(ht))$$

$$\mathsf{tail}(\mathsf{bfl}\, t) = \mathsf{if}\ (\mathsf{isnil}\, (\mathsf{bfl}\, t))\ \mathsf{then\ error\ else}\ \mathsf{inr}\, \mathsf{bfl}(tt * \mathsf{lsb}(\mathsf{snd}(ht)))$$

where $\mathsf{head}\, t = \mathsf{inr}\, ht$, $\mathsf{tail}\, t = \mathsf{inr}\, tt$ and $*$ denotes append of lists. If $t = \mathsf{nil}$, we simply define $\mathsf{bfl}\ \mathsf{nil} = \mathsf{nil}$. The function $\mathsf{bfl}$ is programmed by coiteration as

$$\mathsf{bfl} = \lambda x.\mathsf{CoIt}_2(\mathsf{map}_1, \mathsf{map}_2, s_1, s_2, x)$$

where

- $\mathsf{map}_1 = \lambda f \lambda x.x$, $\mathsf{map}_2 = \lambda f \lambda x.\mathsf{case}(x, y.\, \mathsf{inl}\, y, z.\, \mathsf{inr}\, f z)$
- $s_1 : \mathsf{List}(A \times \mathsf{BLTree}\, A) \to 1 + A$

$$s_1 = \lambda w.\mathsf{if}\ (\mathsf{isnil}\, w)\ \mathsf{then\ error\ else}\ \mathsf{inr}(\mathsf{fst}(\mathsf{head}\, w))$$

- $s_2 : \mathsf{List}(A \times \mathsf{BLTree}\, A) \to 1 + \mathsf{List}(A \times \mathsf{BLTree}\, A)$

$$s_2 = \lambda w.\mathsf{if}\ (\mathsf{isnil}\, w)\ \mathsf{then\ error\ else}\ \mathsf{inr}(\mathsf{tail}\, w * \mathsf{lsb}(\mathsf{snd}(\mathsf{head}\, w))).$$

- Depth first search: this is easily implemented as a function $\mathsf{dfs} : \mathsf{BLTree}\, A \to A^{\infty}$ defined as $\mathsf{dfs} = \mathsf{dfl} \circ \mathsf{lsb}$, where $\mathsf{dfl}$ is obtained from $\mathsf{bfl}$ by changing the second step function to:
  $s_2 = \lambda w.\mathsf{if}\ (\mathsf{isnil}\, w)\ \mathsf{then\ error\ else}\ \mathsf{inr}(\mathsf{lsb}(\mathsf{snd}(\mathsf{head}\, w)) * (\mathsf{tail}\, w))$.

The above examples show the advantage of using clauses. In particular we make emphasis in the direct definition of constructors or destructors which generally avoids the use of injections or projections. This feature also modularizes the definition of monotonicity witnesses and the mechanism of (co)inductive definitions by (co)iteration, (co)recursion or inversion.

### 5.4. MCICT is safe

We prove safety of the system in a strong way by proving its termination by means of an embedding of MCICT into the already terminating system MICT. Moreover, for the type-preservation (subject- reduction) property we provide here a direct proof.

#### 5.4.1. *Strong normalization of* MCICT

The strong normalization of the clausular system MCICT will follow the standard technique of type-respecting, reduction-preserving translations or embeddings, see [20]. This time an embedding $(\cdot)'$, into the system MICT will be

given. The main idea is to define $\mu X(A_1,\ldots,A_k)'$ as $\mu X.A'_1 + \ldots + A'_k$ and $\nu X(A_1,\ldots,A_k)'$ as $\nu X.A'_1 \times \ldots \times A'_k$. Some details are given below.

From now on we agree to associate sum and product to the right.

**Definition 5.1.** The following syntax sugar will be useful, where $k \geq 2$:

$$\begin{array}{rcl} \mathsf{inj}_j^k & = & \lambda z.\, \mathsf{inr}^{j-1}(\mathsf{inl}\, z), \;\; 1 \leq j < k \\ \mathsf{inj}_k^k & = & \lambda z.\, \mathsf{inr}^{k-1}\, z \\ \pi_{k,j} & = & \lambda z.\, \mathsf{fst}(\mathsf{snd}^{j-1}\, z), \;\; 1 \leq j < k \\ \pi_{k,k} & = & \lambda z.\, \mathsf{snd}^{k-1}\, z. \end{array}$$

These are, of course, the canonical injections and projections for a $k$-sum and $k$-product.

Next we define some terms that will be needed for the embedding of (co)iterators, (co)recursors and $\mathsf{in}, \mathsf{out}$ functions.

**Definition 5.2** (MICT)**.** Given variables $x_1,\ldots,x_k,y_1,\ldots,y_k,f,u,v,w,z$ we define, for $k \geq 2$ and $1 \leq j \leq k$, the following families of terms $t_j, r_j, q_j, p_j$:

$$\begin{array}{rcl} t_j[u] & = & \mathsf{inj}_j^k(x_j f u) \quad 1 \leq j \leq k \\ \\ r_0[v] & = & t_k[v] \\ r_{j+1}[v] & = & \mathsf{case}(v, x.t_{k-(j+1)}[x], y.r_j[y]) \quad 0 \leq j < k-1 \\ \\ q_0[w] & = & y_k w \\ q_{j+1}[w] & = & \mathsf{case}(w, x.y_{k-(j+1)}x, y.q_j[y]) \quad 0 \leq j < k-1 \\ \\ p_j[z] & = & x_j f(\pi_{k,j} z) \quad 1 \leq j \leq k. \end{array}$$

Observe that the free variables are:

$$\begin{array}{rcl} FV(t_j[u]) & = & \{x_j, f, u\} \\ FV(r_j[v]) & = & \{x_{k-j},\ldots,x_k, f, v\} \\ FV(q_j[w]) & = & \{y_{k-j},\ldots,y_k, w\} \\ FV(p_j[z]) & = & \{x_j, f, z\}. \end{array}$$

**Definition 5.3.** Given variables $\vec{x},\vec{y}$ with $|\vec{x}| = |\vec{y}| = k$, we define the following terms:

$$\begin{array}{rcl} \mathcal{M}^+[\vec{x}] & = & \lambda f \lambda z.r_{k-1}[z] \\ \mathcal{S}^+[\vec{y}] & = & \lambda w.q_{k-1}[w] \\ \mathcal{M}^\times[\vec{x}] & = & \lambda f.\lambda z.\langle p_1[z],\ldots,p_k[z]\rangle \\ \mathcal{S}^\times[\vec{y}] & = & \lambda w.\langle y_1 w,\ldots,y_k w\rangle. \end{array}$$

Observe that

$$FV(\mathcal{M}^+[\vec{x}]) = FV(\mathcal{M}^\times[\vec{x}]) = \vec{x}$$
$$FV(\mathcal{S}^+[\vec{y}]) = FV(\mathcal{S}^\times[\vec{y}]) = \vec{y}.$$

Now we are in a position to define the translation.

**Definition 5.4.** The embedding $(\cdot)' : \mathsf{MCICT} \to \mathsf{MICT}$ is defined in two parts, first we define it for the degenerate cases of (co)inductive types without clauses $\mu X(), \nu X()$, which are special encoded types. Then we give the general definition which excludes the previous cases. The omitted cases are just homomorphic[2]

$$\begin{array}{rcl}
(\mu X())' & = & \forall X X \\
\mathsf{It}_0(t)' & = & t' \\
\mathsf{Rec}_0(t)' & = & t' \\
(\nu X())' & = & 1+1 \\
\mathsf{CoIt}_0(t)' & = & \mathsf{inl}\star \\
\mathsf{CoRec}_0(t)' & = & \mathsf{inr}\star
\end{array}$$

Next, the general definition where $k \geq 1$

$$\begin{array}{rcl}
\big(\mu X(F_1,\ldots,F_k)\big)' & = & \mu X.F_1' + \ldots + F_k' \\
\big(\nu X(F_1,\ldots,F_k)\big)' & = & \nu X.F_1' \times \ldots \times F_k' \\
x' & = & x \\
\mathsf{in}_{1,1}\, t' & = & \mathsf{in}\, t' \\
\mathsf{in}_{k,i}\, t' & = & \mathsf{in}(\mathsf{inj}_i^k\, t')\ k \geq 2 \\
\mathsf{It}_1(m,s,t)' & = & \mathsf{It}(m',s',t') \\
\mathsf{It}_k(\vec{m},\vec{s},t)' & = & \mathsf{It}(\mathcal{M}^+[\vec{m}'],\mathcal{S}^+[\vec{s}\,'],t') \quad k \geq 2 \\
\mathsf{Rec}_1(m,s,t)' & = & \mathsf{Rec}(m',s',t') \\
\mathsf{Rec}_k(\vec{m},\vec{s},t)' & = & \mathsf{Rec}(\mathcal{M}^+[\vec{m}'],\mathcal{S}^+[\vec{s}\,'],t') \quad k \geq 2 \\
(\mathsf{out}_{1,1}\, t)' & = & \mathsf{out}\, t' \\
(\mathsf{out}_{k,i}\, t)' & = & \pi_{k,i}(\mathsf{out}\, t')\ k \geq 2 \\
\mathsf{out}_k^{-1}(\vec{m},\vec{t}\,)' & = & \mathsf{out}^{-1}(\mathcal{M}^\times[\vec{m}'],\langle t_1',\ldots,t_k'\rangle) \\
\mathsf{CoIt}_1(m,s,t)' & = & \mathsf{CoIt}(m',s',t') \\
\mathsf{CoIt}_k(\vec{m},\vec{s},t)' & = & \mathsf{CoIt}(\mathcal{M}^\times[\vec{m}'],\mathcal{S}^\times[\vec{s}\,'],t') \quad k \geq 2 \\
\mathsf{CoRec}_1(m,s,t)' & = & \mathsf{CoRec}(m',s',t') \\
\mathsf{CoRec}_k(\vec{m},\vec{s},t)' & = & \mathsf{CoRec}(\mathcal{M}^\times[\vec{m}'],\mathcal{S}^\times[\vec{s}\,'],t') \quad k \geq 2
\end{array}$$

where the terms $\mathcal{M}^+, \mathcal{M}^\times, \mathcal{S}^+, \mathcal{S}^\times$ are taken from Definition 5.3.

The proofs of type-respect and reduction preservation are routinary.

**Proposition 5.1** (type respect)**.** *If* $\Gamma \vdash_{\mathsf{MCICT}} r : B$ *then* $\Gamma' \vdash_{\mathsf{MICT}} r' : B'$*, where for* $\Gamma = \{x_1 : A_1, \ldots, x_n : A_n\}$ *we put* $\Gamma' = \{x_1 : A_1', \ldots, x_n : A_n'\}$*.*

*Proof.* Induction on $\vdash_{\mathsf{MCICT}}$. □

**Proposition 5.2** (preservation of reduction)**.** *If* $r \to_\beta s$ *in* MCICT *then* $r' \to_\beta^+ s'$ *in* MICT*.*

*Proof.* Induction on $\to_\beta$ in MCICT. □

**Proposition 5.3.** MCICT *is strongly normalizing.*

*Proof.* Immediate from Corollary 4.2 and Proposition 5.2. □

[2] V.gr. $X' = X$, $(A \to B)' = A' \to B'$, $(\forall X.A)' = \forall X.A'$, etc.

### 5.5. Type preservation for MCICT

In this section we give a detailed proof of type preservation (subject reduction) for the system MCICT. This important property is not trivial to prove due to the absence of type annotations in terms given by the Curry-style presentation and to the presence of untraceable typing rules, which are rules whose application cannot be traced by simply looking at terms only. This part of our work is completely new with respect to the related systems of [19,20], due to the fact that on those works the presentation is in Church-style, a feature which yields subject reduction trivially. Our proof is based on Krivine's proof for system F (see [17]).

**Definition 5.5.** Given a type $A$ and a context $\Gamma$ we define the set $\mathcal{C}_\Gamma(A)$ of $\Gamma$-instances of $A$ as the least class of types such that the following conditions hold

(I1) $A \in \mathcal{C}_\Gamma(A)$.
(I2) If $B \in \mathcal{C}_\Gamma(A)$ and $X \notin FV(\Gamma)$ then $B[X := F] \in \mathcal{C}_\Gamma(A)$, for every type $F$.

**Lemma 5.1.** *If $X \notin FV(\Gamma)$ then $\mathcal{C}_\Gamma(A[X := F]) \subseteq \mathcal{C}_\Gamma(A)$.*

*Proof.* By condition (I1) of Definition 5.5, $A \in \mathcal{C}_\Gamma(A)$ which implies, as $X \notin FV(\Gamma)$, that $A[X := F] \in \mathcal{C}_\Gamma(A)$. The claim follows now by the minimality of $\mathcal{C}_\Gamma(A[X := F])$. □

**Definition 5.6.** A type $A$ is an open type if it is not a universal quantification. The interior of a type $A$, denoted $A^\circ$ is defined as follows:

$$\begin{array}{rcll} A^\circ & = & A, & \text{if } A \text{ is open.} \\ (\forall X A)^\circ & = & A^\circ. & \end{array}$$

**Definition 5.7.** We say that an inference rule is non-traceable if its application is not reflected in the type system. That is, if the term in the conclusion equals one of the terms of the premisses. Otherwise the rule is called traceable.

In MCICT the non-traceable rules are the two rules for universal quantification.

The next lemma is central to fulfill our goal.

**Lemma 5.2** (main lemma)**.** *Let $\widetilde{A}$ be an open type. If $\Gamma \vdash t : \widetilde{A}$ is derived from $\Gamma \vdash t : A$ using only non-traceable rules, then $\widetilde{A} \in \mathcal{C}_\Gamma(A^\circ)$.*

*Proof.* Induction on the number of steps in the derivation of $\Gamma \vdash t : \widetilde{A}$ from $\Gamma \vdash t : A$. We perform a case analysis on the first rule used in that derivation.

- $(\forall I)$. We have $\Gamma \vdash t : \widetilde{A}$ from $\Gamma \vdash t : \forall X A$ where $X \notin FV(\Gamma)$, therefore by IH we get $\widetilde{A} \in \mathcal{C}_\Gamma((\forall X A)^\circ)$. But $(\forall X A)^\circ = A^\circ$ therefore $\widetilde{A} \in \mathcal{C}_\Gamma(A^\circ)$.
- $(\forall E)$. We have $A = \forall X \forall Y_1 \ldots \forall Y_n B$ with $B$ open, *i.e.*, $A^\circ = B = B^\circ$ and after the application of $(\forall E)$ we get $\Gamma \vdash t : B[X := F]$. By IH we have $\widetilde{A} \in \mathcal{C}_\Gamma\big((\forall Y_1 \ldots \forall Y_n . B[X := F])^\circ\big) = \mathcal{C}_\Gamma(B[X := F]^\circ)$. We have two subcases:

– $B^\circ \neq X$. In this case $B[X := F]$ is open and of the same syntactic form as $B$. The I.H. yields $\widetilde{A} \in \mathcal{C}_\Gamma(B[X := F])$ and assuming w.l.o.g. that $X \notin FV(\Gamma)$, Lemma 5.1 implies that $\widetilde{A} \in \mathcal{C}_\Gamma(B)$. That is, $\widetilde{A} \in \mathcal{C}_\Gamma(A^\circ)$.

– $B^\circ = X$. In this case $B[X := F] = F$ and the IH yields $\widetilde{A} \in \mathcal{C}_\Gamma(B[X := F]^\circ) = \mathcal{C}_\Gamma(F^\circ) = \mathcal{C}_\Gamma(B[X := F^\circ])$, which implies by Lemma 5.1, assuming w.l.o.g. that $X \notin FV(\Gamma)$, that $\widetilde{A} \in \mathcal{C}_\Gamma(B)$. That is, $\widetilde{A} \in \mathcal{C}_\Gamma(A^\circ)$. □

Next, we classify terms according to the kind of rule used to build them.

**Definition 5.8.** A term $t$ is called an $I$-term if it was generated by an introduction rule, *i.e.*, $I$-terms are terms of the following shapes:

$$\lambda x.r,\ \langle r, s\rangle,\ \mathsf{inl}\, r,\ \mathsf{inr}\, s, \mathsf{in}_{k,j}\, r,\ \mathsf{CoIt}_k(\vec{m}, \vec{s}, r),\ \mathsf{CoRec}_k(\vec{m}, \vec{s}, r),\ \mathsf{out}_k^{-1}(\vec{m}, \vec{r}).$$

Analogously $E$-terms are those generated by an elimination rule, *i.e.* they are terms of the following shapes:

$$rs,\ \mathsf{fst}\, r,\ \mathsf{snd}\, r,\ \mathsf{case}(r, x.s, y.t),\ \mathsf{It}_k(\vec{m}, \vec{s}, r),\ \mathsf{Rec}_k(\vec{m}, \vec{s}, r),\ \mathsf{out}_{k,j}\, r.$$

Observe that every term is either a variable, an $I$-term, or an $E$-term. The next lemma characterizes derivations of open types according to this classification.

**Lemma 5.3** (generation lemma)**.** *If $\Gamma \vdash t : A$, where $A$ is an open type, then:*

- *If $t$ is the variable $x$, then there exists a declaration $(x : B) \in \Gamma$ such that $A \in \mathcal{C}_\Gamma(B^\circ)$.*
- *If $t$ is an $I$-term, then $\Gamma \vdash t : A$ is the conclusion of an instance of the rule generating $t$.*
- *if $t$ is an $E$-term, then there exists a type $B$ such that $\Gamma \vdash t : B$ is the conclusion of the rule generating $t$ and $A \in \mathcal{C}_\Gamma(B^\circ)$.*

*Proof.* Let us consider in the derivation of $\Gamma \vdash t : A$ the last step where a traceable rule $\mathcal{R}$ occurs, thus $\mathcal{R}$ is the rule generating $t$. Assuming that the conclusion of $\mathcal{R}$ is $\Gamma \vdash t : B$ the main Lemma 5.2 implies that $A \in \mathcal{C}_\Gamma(B^\circ)$. We perform a case analysis on $t$.

- $t = x$. Then $\mathcal{R}$ is $(Var)$ and therefore a declaration $(x : B) \in \Gamma$ exists, and as mentioned before $A \in \mathcal{C}_\Gamma(B^\circ)$.
- $t$ is an $E$-term. This case is immediate as $\mathcal{R}$ is the rule generating $t$.
- $t$ is an $I$-term. Let us perform a case analysis on the syntactic shape of $t$. We focus on the case $t = \mathsf{in}_{k,j}\, r$. Therefore the rule $\mathcal{R}$ is $(\mu I)$, $B = \mu Y(C_1, \ldots, C_\ell)$ and $\Gamma \vdash r : C_j[Y := \mu Y(C_1, \ldots, C_\ell)]$. Clearly $B = B^\circ$ implies $A \in \mathcal{C}_\Gamma(B)$. Let us define a set $\mathcal{C}$ as

$$\mathcal{C} = \big\{\, \mu X(D_1, \ldots, D_k) \mid \Gamma \vdash r : D_j[X := \mu X(D_1, \ldots, D_k)], \text{ for some } k, \text{ and } D_j \,\big\}.$$

We claim that $\mathcal{C}_\Gamma(B) \subseteq \mathcal{C}$. Let us prove that the properties (I1) and (I2) of Definition 5.5 hold for $\mathcal{C}$.
(I1) Obviously $B \in \mathcal{C}$.
(I2) Assume $R = \mu X(D_1, \ldots, D_k) \in \mathcal{C}$ and $Z \notin FV(\Gamma)$. We have

$$\begin{aligned} R[Z := F] \;&=\; \mu X(D_1, \ldots, D_k)[Z := F] \\ &= \mu X\big(D_1[Z := F], \ldots, D_k[Z := K]\big). \end{aligned}$$

$R \in \mathcal{C}$ implies $\Gamma \vdash r : D_j[X := \mu X(D_1, \ldots, D_k)]$. From this, as $Z \notin FV(\Gamma)$ we can build a derivation of

$$\Gamma \vdash r : D_j[X := \mu X(D_1, \ldots, D_k)][Z := F].$$

Finally, using substitution properties, we obtain

$$\Gamma \vdash r : D_j[Z := F]\big[X := \mu X(D_1, \ldots, D_k)[Z := F]\big].$$

But

$$\begin{aligned} &D_j[Z := F]\big[X := \mu X(D_1, \ldots, D_k)[Z := F]\big] = \\ &D_j[Z := F]\big[X := \mu X(D_1[Z := F], \ldots, D_k[Z := F])\big] \end{aligned}$$

which yields $R[Z := F] \in \mathcal{C}$.
Therefore by minimality of $\mathcal{C}_\Gamma(B)$ we conclude $\mathcal{C}_\Gamma(B) \subseteq \mathcal{C}$, which yields $A \in \mathcal{C}$. Finally observe that the definition of $\mathcal{C}$ implies that $\Gamma \vdash t : A$ is the conclusion of the rule $(\mu I)$ which is the rule generating $t$, as desired. The other cases for an $I$-term are proved analogously. □

We are now ready to prove the type-preservation of system MCICT.

**Proposition 5.4** (one-step subject reduction)**.** *If* $\Gamma \vdash t : A$ *and* $t \to_\beta \widehat{t}$ *in one step, then* $\Gamma \vdash \widehat{t} : A$.

*Proof.* Induction on $\vdash$. The case for rule $(Var)$ is immediate since there is no redex involved. For an introduction rule the proof is immediate from the IH. For elimination rules the proof is direct, as an example we show it for the rule $(\nu E)$.

We have $A = F_j[X := \nu X(F_1, \ldots, F_k)]$ and $t = \mathsf{out}_{k,j}\, s$ with $\Gamma \vdash t : A$ concluded from $\Gamma \vdash s : \nu X(F_1, \ldots, F_k)$.

The interesting subcases are $s = \mathsf{CoIt}_k(\vec{m}, \vec{s}, r), s = \mathsf{CoRec}_k(\vec{m}, \vec{s}, r)$ and $s = \mathsf{out}^{-1}(\vec{m}, \vec{r})$. We develop the proof for $s = \mathsf{CoIt}_k(\vec{m}, \vec{s}, r)$ and $\widehat{t} = m_j\big(\lambda z.\mathsf{CoIt}_k(\vec{m}, \vec{s}, z)\big)(s_j r)$. From the assumption $\Gamma \vdash \mathsf{CoIt}_k(\vec{m}, \vec{s}, r) : \nu X(F_1, \ldots, F_k)$ the generation Lemma 5.3 yields $\Gamma \vdash m_i : F_i\, \mathsf{mon}\, X$, $\Gamma \vdash s_i : B \to F_i[X := B]$ for all $1 \le i \le k$, and $\Gamma \vdash r : B$. It is easy to see that $\Gamma \vdash \lambda z.\mathsf{CoIt}_k(\vec{m}, \vec{s}, z) : B \to \nu X(F_1, \ldots, F_k)$, which implies $\Gamma \vdash m_j\big(\lambda z.\mathsf{CoIt}_k(\vec{m}, \vec{s}, z)\big) : F_j[X := B] \to F_j[X := \nu X(F_1, \ldots, F_k)]$.

On the other hand we have $\Gamma \vdash s_j r : F_j[X := B]$. Therefore $\Gamma \vdash m_j\big(\lambda z.\mathsf{Colt}_k(\vec{m}, \vec{s}, z)\big)(s_j r) : F_j[X := \nu X(F_1, \ldots, F_k)]$, that is, $\Gamma \vdash \widehat{t} : A$. □

This proposition yields subject reduction in the general case as a corollary.

**Corollary 5.1** (subject reduction for MCICT)**.** *If* $\Gamma \vdash r : A$ *and* $r \to_\beta \widehat{r}$ *then* $\Gamma \vdash \widehat{r} : A$.

*Proof.* Induction on the length of the reduction sequence $r \to_\beta \widehat{r}$. □

This finishes the proof of type safety for our system MCICT.

## 6. Conclusions

In this paper we have presented two Curry-style extensions, called MICT and MCICT, of system F with monotone (co)inductive types and including not only (co)iteration but also primitive (co)recursion and coinductive inversion principles. Both systems are proved to be strongly normalizing; in the case of MICT by a non-trivial extension of saturated sets for coinductive types, whereas for MCICT, by a straightforward embedding into MICT. About subject-reduction (type preservation), we observe that this property is not trivial for Curry-style systems, and prove it in detail by extending Krivine's method for system F. Therefore, our systems provide a framework with native forms of recursion suitable to be implemented within the total functional programming paradigm. The expressiveness of our systems is made explicit by means of several programming examples; a comparison of the programming methodologies allows us to conclude that the clausular feature of system MCICT makes it more adequate to implementation since it allows the use of several constructors/destructors as in the usual functional programming languages and it is therefore more friendly to the user as it modularizes definitions of functions and monotonicity witnesses.

### 6.1. Related work

The system MICT can be thought of as an extension of a Curry-style version of the system IMIT of *introduction-based monotone inductive types*, developed in [19] with coinductive types, coiteration, primitive corecursion and inversion principles. On the other hand, the higher-order system $\underline{\mathsf{It}}^\omega$ of [1] can be seen as an extension of the (co)iterative fragment of our first system MICT within the framework of higher-order parametric polymorphism. Some other related extensions in Church-style are studied in [7,12,20]. The system MCICT can be seen as an extension of Hagino's categorical type system $\lambda C$ (see [13,40]), by means of parametric polymorphism, primitive (co)recursion and inversion principles. More recently and following the same line of research we have proposed some extensions of MICT with course-of-value iteration principles corresponding to a categorical combinator called histomorphism, see [28].

Our systems have a counterpart also under the Curry-Howard correspondence, as logics of (co)inductive definitions, developed in [27]. These logics are obtained by extending the second-order logic AF2 and are also related to the work in [37]; their main application is to provide a framework to extract programs from proofs à la Krivine-Parigot (see [29]), using as the underlying programming language the formalisms of this article. Again, this has been done in [27], where we have also developed Mendler-style systems of (co)inductive types. These kind of systems have a strong expressive power and naturally arise in higher-order functional programming (see [1]).

### 6.2. Future work

Above all, it is desirable to implement our systems. A starting point in this direction is provided by the implementations in Haskell given in [38]. On the other hand we are interested in the expansion of our formalisms with more (co)recursion schemes, hopefully within the total functional programming paradigm, corresponding to categorical combinators such as hylomorphisms (see [4,24]) or the more recent scheme of [16], called dynamorphism.

*Acknowledgements.* The author thanks an anonymous referee for the careful reading of the original manuscript, and the valuable suggestions that deeply improved the presentation of the paper, specially the examples in Sections 3.2 and 5.3, and the proof of strong normalization given in Section 4. I am also very thankful to Martha Elena Buschbeck-Alvarado for improving the English manuscript.

## Appendix A: Syntax sugar

Future work includes the implementation of our language MCICT. Here we propose some syntax sugar to avoid the heavy use of $\lambda$-terms.

- Data type definition. An inductive type $\mu X(F_1, \ldots, F_n)$ can be declared as:

$$\begin{array}{rcl} < type\_name > & = & \texttt{Inductive}\ X\ \texttt{with constructors} \\ & & c_1 : F_1 \to X \\ & & \vdots \\ & & c_n : F_n \to X \end{array}$$

  where $c_i$ are particular names for the constructors, $c_i = \mathsf{in}_{k,i}$.

- Codatatype definition. A Coinductive type $\nu X(F_1, \ldots, F_n)$ is declared as:

$$\begin{array}{rcl} < type\_name > & = & \texttt{Coinductive}\ X\ \texttt{with destructors} \\ & & \quad d_1 : X \to F_1 \\ & & \quad \vdots \\ & & \quad d_n : X \to F_n \end{array}$$

  where $d_i$ are particular names for the destructors, $d_i = \mathsf{out}_{k,i}$.

- A function $\mathsf{fun} : \mathcal{I} \to B$, where $\mathcal{I} = \mu X(F_1, \ldots, F_k)$, defined by iteration $\mathsf{fun} = \lambda x.\mathsf{It}_k(\vec{m}, \vec{s}, x)$ is programmed as:

$$\begin{array}{rcl} \mathsf{fun} & = & \texttt{Iterator of}\ \mathcal{I}\ \texttt{to}\ B\ \texttt{with steps} \\ & & \quad s_1 : F_1[X := B] \to B \\ & & \quad \vdots \\ & & \quad s_k : F_k[X := B] \to B \\ & & \texttt{where}\ \mathsf{map}_1 = m_1, \ldots \mathsf{map}_k = m_k. \end{array}$$

  Analogously if the definition is by recursion, say $\mathsf{fun} = \lambda x.\mathsf{Rec}_k(\vec{m}, \vec{s}, x)$, we adjust the types of the constructors and change the word "`Iterator`" to "`Recursor`".

- A function $\mathsf{fun} : B \to \mathcal{C}$, where $\mathcal{C} = \nu X(F_1, \ldots, F_k)$, defined by corecursion $\mathsf{fun} = \lambda x.\mathsf{CoRec}_k(\vec{m}, \vec{s}, x)$ is represented as:

$$\begin{array}{rcl} \mathsf{fun} & = & \texttt{Corecursor of}\ \mathcal{C}\ \texttt{from}\ B\ \texttt{with steps} \\ & & \quad s_1 : B \to F_1[X := B] \\ & & \quad \vdots \\ & & \quad s_k : B \to F_k[X := B] \\ & & \texttt{where}\ \mathsf{map}_1 = m_1, \ldots \mathsf{map}_k = m_k. \end{array}$$

  Analogously if the definition is by coiteration, say $\mathsf{fun} = \lambda x.\mathsf{CoIt}_k(\vec{m}, \vec{s}, x)$, we adjust the types of the destructors and change the word "`Corecursor`" to "`Coiterator`".

- After the syntax sugar, the operational semantics looks as follows:
  - If $\mathsf{fun}$ is defined by iteration then $\mathsf{fun}(c_i x) \to s_i\big(\mathsf{map}_i\ \mathsf{fun}\ x\big)$
  - If $\mathsf{fun}$ is defined by recursion then $\mathsf{fun}(c_i x) \to s_i\big(\mathsf{map}_i\ \langle \mathsf{Id}, \mathsf{fun}\rangle\ x\big)$
  - If $\mathsf{fun}$ is defined by corecursion then $d_i(\mathsf{fun}\ x) \to \mathsf{map}_i\ [\mathsf{Id}, \mathsf{fun}]\ (s_i x)$
  - If $\mathsf{fun}$ is defined by coiteration then $d_i(\mathsf{fun}\ x) \to \mathsf{map}_i\ \mathsf{fun}\ (s_i x)$.

## Appendix B: Canonical monotonicity witnesses

In this appendix we present a canonical selection of monotonicity witnesses for system MCICT, which essentially corresponds to the usual definitions for the positive cases. We do not restrict ourselves to strict positivity and define also antimonotonicity witnesses corresponding to contravariant functors. Moreover, we define witnesses for interleaving types.

**Definition B.1** (antimonotonicity)**.** Given a type $F$ and a type variable $X$, we define the type $F\,\mathsf{mon}^-\,X$ as:

$$F\,\mathsf{mon}^-\,X = \forall X.\forall Y.(X \to Y) \to F[X := Y] \to F.$$

If a term $m$ inhabits the type $F\,\mathsf{mon}^-\,X$ in a given context, then the syntactic functor $\langle \lambda X.F, m\rangle$ will be antimonotone (contravariant) in the same context.

**Definition B.2** (generic (anti)monotonicity witnesses)**.** We define the following MCICT-terms:

- $\mathcal{M}_{\mathsf{id}} = \lambda x.x.$
- $\mathcal{M}_{\mathsf{triv}} = \lambda f \lambda x.x.$
- $\mathcal{M}_{\to} = \lambda m_1 \lambda m_2 \lambda f \lambda x \lambda y.m_2 f(x(m_1 f y)).$
- $\mathcal{M}_{\forall} = \lambda m \lambda f \lambda x.m f x.$
- $\mathcal{M}_{\times} = \lambda m_1 \lambda m_2 \lambda f \lambda x.\langle m_1 f(\mathsf{fst}\,x), m_2 f(\mathsf{snd}\,x)\rangle.$
- $\mathcal{M}_{+} = \lambda m_1 \lambda m_2 \lambda f \lambda x.\mathsf{case}(x, y.\,\mathsf{inl}\,m_1 f y, z.\,\mathsf{inr}\,m_2 f z).$
- $\mathcal{M}^k_{\mu} = \lambda \vec{m} \lambda \vec{n} \lambda f \lambda x.\mathsf{It}_k(\vec{m}, \vec{s}, x)$, where $s_i = \lambda z.\,\mathsf{in}_{k,i}(n_i f z)$, for all $1 \le i \le k$.
- $\mathcal{M}^k_{\nu} = \lambda \vec{m} \lambda \vec{n} \lambda f \lambda x.\,\mathsf{out}_k^{-1}(\vec{m}, \vec{s})$ where $s_i = n_i f(\mathsf{out}_{k,i}\,x)$, for all $1 \le i \le k$.

**Proposition B.1** (derived typing rules for (anti)monotonicity)**.** *The following rules are derivable:*

- $\Gamma \vdash \mathcal{M}_{\mathsf{id}} : X\,\mathsf{mon}\,X.$
- *If* $X \notin FV(F)$ *then* $\Gamma \vdash \mathcal{M}_{\mathsf{triv}} : F\,\mathsf{mon}\,X$ *and* $\Gamma \vdash \mathcal{M}_{\mathsf{triv}} : F\,\mathsf{mon}^-\,X.$
- *If* $\Gamma \vdash m_1 : F\,\mathsf{mon}^-\,X$ *and* $\Gamma \vdash m_2 : G\,\mathsf{mon}\,X$*, then*

$$\Gamma \vdash \mathcal{M}_{\to} m_1 m_2 : (F \to G)\,\mathsf{mon}\,X.$$

- *If* $\Gamma \vdash m_1 : F\,\mathsf{mon}\,X$ *and* $\Gamma \vdash m_2 : G\,\mathsf{mon}^-\,X$*, then*

$$\Gamma \vdash \mathcal{M}_{\to} m_1 m_2 : (F \to G)\,\mathsf{mon}^-\,X.$$

- *If* $\Gamma \vdash t : \forall Z.F\,\mathsf{mon}\,X$*, then* $\Gamma \vdash \mathcal{M}_{\forall} t : (\forall Z.F)\,\mathsf{mon}\,X.$
- *If* $\Gamma \vdash t : \forall Z.F\,\mathsf{mon}^-\,X$*, then* $\Gamma \vdash \mathcal{M}_{\forall} t : (\forall Z.F)\,\mathsf{mon}^-\,X.$
- *If* $\Gamma \vdash m_1 : F\,\mathsf{mon}\,X$ *and* $\Gamma \vdash m_2 : G\,\mathsf{mon}\,X$*, then*

$$\Gamma \vdash \mathcal{M}_{\times} m_1 m_2 : (F \times G)\,\mathsf{mon}\,X.$$

- *If* $\Gamma \vdash m_1 : F\,\mathsf{mon}^-\,X$ *and* $\Gamma \vdash m_2 : G\,\mathsf{mon}^-\,X$*, then*

$$\Gamma \vdash \mathcal{M}_{\times} m_1 m_2 : (F \times G)\,\mathsf{mon}^-\,X.$$

- *If* $\Gamma \vdash m_1 : F \,\mathsf{mon}\, X$ *and* $\Gamma \vdash m_2 : G \,\mathsf{mon}\, X$, *then*

$$\Gamma \vdash \mathcal{M}_+ m_1 m_2 : (F + G) \,\mathsf{mon}\, X.$$

- *If* $\Gamma \vdash m_1 : F \,\mathsf{mon}^- X$ *and* $\Gamma \vdash m_2 : G \,\mathsf{mon}^- X$, *then*

$$\Gamma \vdash \mathcal{M}_+ m_1 m_2 : (F + G) \,\mathsf{mon}^- X.$$

- *If* $\Gamma \vdash m_i : (\forall X.F_i \,\mathsf{mon}\, Z)$ *and* $\Gamma \vdash n_i : (\forall Z.F_i \,\mathsf{mon}\, X)$, *for all* $1 \leq i \leq k$, *then*

$$\Gamma \vdash \mathcal{M}^k_\mu \vec{m}\vec{n} : \mu Z(F_1, \ldots, F_k) \,\mathsf{mon}\, X.$$

- *If* $\Gamma \vdash m_i : (\forall X.F_i \,\mathsf{mon}\, Z)$ *and* $\Gamma \vdash n_i : (\forall Z.F_i \,\mathsf{mon}^- X)$, *for all* $1 \leq i \leq k$, *then*

$$\Gamma \vdash \mathcal{M}^k_\mu \vec{m}\vec{n} : \mu Z(F_1, \ldots, F_k) \,\mathsf{mon}^- X.$$

- *If* $\Gamma \vdash m_i : (\forall X.F_i \,\mathsf{mon}\, Z)$ *and* $\Gamma \vdash n_i : (\forall Z.F_i \,\mathsf{mon}\, X)$, *for all* $1 \leq i \leq k$, *then*

$$\Gamma \vdash \mathcal{M}^k_\nu \vec{m}\vec{n} : \nu Z(F_1, \ldots, F_k) \,\mathsf{mon}\, X.$$

- *If* $\Gamma \vdash m_i : (\forall X.F_i \,\mathsf{mon}\, Z)$ *and* $\Gamma \vdash n_i : (\forall Z.F_i \,\mathsf{mon}^- X)$, *for all* $1 \leq i \leq k$, *then*

$$\Gamma \vdash \mathcal{M}^k_\nu \vec{m}\vec{n} : \nu Z(F_1, \ldots, F_k) \,\mathsf{mon}^- X.$$

*Proof.* Straightforward □

In particular any witness given in the examples developed in Section 5.3 is called canonical, since its definition and typing is given by the above rules. In an analogous way, we can define canonical monotonicity witnesses and derivable rules for system MICT, such that the witnesses for the examples of Section 3.2 are generated by these rules.

## References


[1] A. Abel, R. Matthes and T. Uustalu, Iteration and coiteration schemes for higher-order and nested datatypes. *Theoret. Comput. Sci.* **333** (2005) 3–66.

[2] C. Böhm and A Berarducci, Automatic synthesis of typed Λ-programs on term algebras. *Theoret. Comput. Sci.* **39** (1985) 135–154.

[3] R.L. Crole, Categories for Types. *Cambridge Mathematical Textbooks*. Cambridge University Press (1993).

[4] M.A. Cunha, Recursion patterns as hylomorphisms. Technical Report DI-PURe-03.11.01, Department of Informatics, University of Minho (2003).

[5] B.A. Davey and H.A. Priestley, *Introduction to Lattices and Order*, 2nd edn. Cambridge University Press (2002).

[6] H. Dybkjær and A. Melton, Comparing Hagino's categorical programming language and typed lambda-calculi. *Theoret. Comput. Sci.* **111** (1991) 145–189.

[7] H. Geuvers, Inductive and coinductive types with iteration and recursion. In *Proceedings of the 1992 workshop on types for proofs and programs*, Båstad, Sweden. Edited by B. Nordström, K. Petterson, G. Plotkin (1992) 183–207. Available via http://www.cs.kun.nl/~herman/BRABasInf_RecTyp.ps.gz.

[8] J. Gibbons and G. Jones, The under-appreciated unfold. In *Proceedings 3rd ACM SIGPLAN international conference on functional programming*, Baltimore, Maryland (1998) 273–279.
[9] J.Y. Girard, Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. Second Scandinavian Logic Symposium. Edited by J.E. Fenstad. North-Holland (1971) 63–92.
[10] J.Y. Girard, Interprétation fonctionnelle et élimination des coupures dans l'arithmetique d'ordre supérieur. Thèse de Doctorat d'État, Université de Paris VII (1972).
[11] J.Y. Girard, Y. Lafont and P. Taylor, Proofs and Types. *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press (1989).
[12] J. Greiner, Programming with inductive and co-inductive types. Technical Report CMU-CS-92-109, Carnegie-Mellon University (1992)
[13] T. Hagino, A typed lambda calculus with categorical type constructors. In *Category Theory and Computer Science*. Edited by D.H. Pitt, A. Poigné and D.E. Rydeheard. *Lect. Notes Comput. Sci.* **283** (1987).
[14] T. Hagino, A Categorical programming language. Ph.D. Thesis CST-47-87 (also published as ECS-LFCS-87-38). Department of Computer Science, University of Edinburgh (1987).
[15] B. Jacobs and J. Rutten, A tutorial on (co)algebras and (co)induction. *EATCS Bull.* **62** (1997) 222–259.
[16] J. Kabanov and V. Vene, Recursion schemes for dynamic programming. In *Proc. 8th Int. Conf. on Mathematics of Program Construction, MPC 06*. Edited by T. Uustalu. *Lect. Notes Comput. Sci.* **4014** (2006) 235–252.
[17] J.L. Krivine, Lambda-calculus, types and models. *Ellis Horwood Series in Computers and their Applications*. Ellis Horwood, Masson (1993).
[18] S. Mac Lane, Categories for the Working Mathematician, 2nd. edn., Vol. **5**. *Graduate Texts in Mathematics*. Springer Verlag (1998).
[19] R. Matthes, Extensions of system F by iteration and primitive recursion on monotone inductive types. Dissertation Universität München (1999). Available via http://www.tcs.informatik.uni-muenchen.de/~matthes/dissertation/matthesdiss.ps.gz
[20] R. Matthes, Monotone (co)inductive types and positive fixed-point types. *RAIRO-Theor. Inf. Appl.* **33** (1999) 309–328.
[21] R. Matthes, Monotone fixed-point types and strong normalization. In *Computer Science Logic*. Edited by G. Gottlob, E. Grandjean, K. Seyr. 12th International Workshop, Brno, Czech Republic, August 24–28, 1998. *Lect. Notes Comput. Sci.* **1584** (1999) 298–312.
[22] R. Matthes, Monotone inductive and coinductive constructors of rank 2. In *Computer Science Logic 2001. Lect. Notes Comput. Sci.* **2142** (2001) 600–614.
[23] R. Matthes, Non-strictly positive fixed-points for classical natural deduction. *Ann. Pure Appl. Logic* **133** (2005) 205–230.
[24] E. Meijer, M. Fokkinga and R. Paterson, Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA 91*. Edited by J. Hughes. *Lect. Notes Comput. Sci.* **523** (1991) 124–144.
[25] N.P. Mendler, Recursive types and type constraints in second-order lambda calculus. In *Proceedings of the 2nd Annual Symposium on Logic in Computer Science*, Ithaca, N.Y. IEEE Computer Society Press, Washington D.C. (1987) 30–36.
[26] N.P. Mendler, Inductive types and type constraints in the second-order lambda calculus. *Ann. Pure Appl. Logic* **51** (1991) 159–172.
[27] F.E. Miranda-Perea, On extensions of AF2 with monotone and clausular (co)inductive definitions. Ph.D. Thesis, Ludwig-Maximilians-Universität München, Germany (2004).
[28] F.E. Miranda-Perea, Some remarks on type systems for course-of-value recursion. In *Proceedings of the third workshop on Logical and Semantic Frameworks with Applications (LSFA 2008)*. Electronic Notes in Theoretical Computer Science **247** (2009).
[29] M. Parigot, Recursive programming with proofs. *Theoret. Comput. Sci.* **94** (1992) 335–356.
[30] E. Poll and J. Zwanenburg, From algebras and coalgebras to dialgebras. In *Coalgebraic Methods in Computer Science (CMCS'2001). Electronic Notes in Theoretical Computer Science* **44** (2001).

[31] J. Reynolds, Towards a theory of type structure. In *Proc. Colloque sur la Programmation*. Edited by B. Robinet. *Lect. Notes Comput. Sci.* **19** (1974).
[32] J.J.M.M. Rutten, Automata and coinduction (an exercise in coalgebra). In *Proceedings of CONCUR '98*. Edited by D. Sangiorgi and R. de Simone. *Lect. Notes Comput. Sci.* **1466** (1998) 194–218.
[33] Z. Spławski and P. Urzyczyn, Type fixpoints: iteration *vs.* recursion. In *Proc. International Conference on Functional Programming*. ACM Press (1999), pp 102–113.
[34] W.W. Tait, A realizability interpretation of the theory of species. In *Logic Colloquium Boston 1971/72*. Edited by R. Parikh. *Lect. Notes Math.* **453** (1975) 240–251.
[35] D. Turner, Total functional programming. *J. Universal Comput. Sci.* **10** (2004) 751–768.
[36] T. Uustalu and V. Vene, Primitive (co)recursion and course-of-value (co)iteration, categorically. *Informatica* **10** (1999) 5–26.
[37] T. Uustalu and V. Vene, Least and greatest fixed-points in intuitionistic natural deduction. *Theoret. Comput. Sci.* **272** (2002) 315–339.
[38] V. Vene, Categorical programming with inductive and coinductive types. *Diss. Math. Univ. Tartuensis* **23**, Univ. Tartu (2000).
[39] P. Wadler, Theorems for free. In *Proc. 4th international conference on functional programming languages and computer architecture*. Imperial College, London (1989), pp 347–359.
[40] G.C. Wraith, A note on categorical datatypes. In *Category Theory and Computer Science*. Edited by D. Pitts *et al. Lect. Notes Comput. Sci.* **389** (1989).

Communicated by G. Longo.
Received February 2nd, 2007. Accepted July 16, 2009.