Representing Agda and Coinduction in the λΠ-calculus modulo rewriting

Thiago Felicissimo

Master 2 Internship from March 2021 to August 2021
Supervised by Frédéric Blanqui and Gilles Dowek
Deducteam/Laboratoire Méthodes Formelles

1 Background

In this section we present the theory on top of which we build our contribution. We start by presenting the $\lambda\Pi$-calculus modulo rewriting (or $\lambda\Pi/\mathcal{R}$-calculus), a logical framework developed and used at Deducteam for expressing logics and checking their proofs. This is followed by a look at coinduction and coinductive types. We then present the proof assistant Agda and detail some particularities of its type system. Finally, we review Agda2Dedukti, a prototype translator for Agda proofs developed by Guillaume Genestier and we detail the features of the proof assistant it is able to handle.

1.1 The $\lambda\Pi$-calculus modulo rewriting

1.1.1 Starting point: a λ-calculus with dependent types

The lambda-calculus with dependent types, or $\lambda\Pi$-calculus, was proposed by [8] as a logical framework in which many proof and type systems can be expressed. Its syntax is given by

$$A, B, M, N ::= x \in \mathcal{X} \mid c \in \mathcal{C} \mid \textbf{Type} \mid \textbf{Kind} \mid MN \mid \lambda x : A.M \mid \Pi x : A.B$$

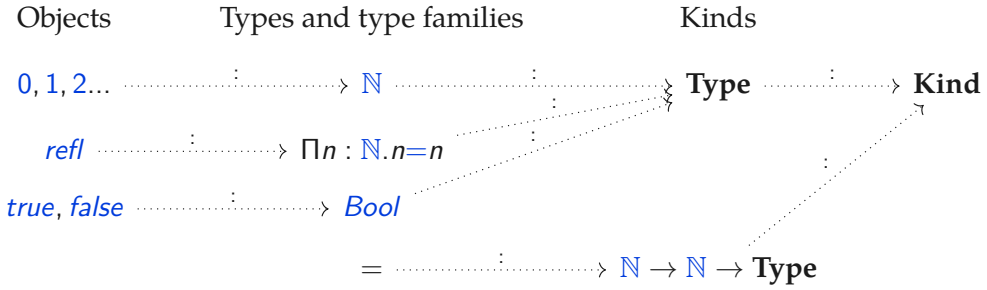
where $\mathcal{C}$ is an infinite set of constants and $\mathcal{X}$ is an infinite set of variables. We denote $\Lambda_{\lambda\Pi}$ the set of terms generated by this grammar. Conversion is defined as usual by β-equivalence, and we write $\Pi x : A.B$ as $A \to B$ when x does not appear in B.

A *context* Γ is a finite sequence of pairs $x : A$, where x is a variable and $A \in \Lambda_{\lambda\Pi}$, such that any variable can only appear once. A *signature* Σ is a finite sequence of pairs $c : A$, where c is a constant, $A \in \Lambda_{\lambda\Pi}$ and every constant can only appear once. As the declaration of constants in Σ is done all the time when declaring encodings of theories, we will write them in blue to explicit the fact that they are added to the signature Σ. Typing in the $\lambda\Pi$-calculus is defined through judgments of the form $\Sigma; \Gamma \vdash M : A$, for $M, A \in \Lambda_{\lambda\Pi}$. We refer to the appendix A for the typing rules.

Intuitively, most terms can be separated into types (terms typed by **Type**) and objects (terms typed by a type object). **Type** is the type of all the object types, whereas **Kind** is there mostly for "administrative" reasons[1]. For instance, if we want to have a symbol $\mathbb{N}$ to represent natural numbers and a constant 0 to represent the number zero, the only way to have $0 : \mathbb{N}$ is by declaring $\mathbb{N} : \textbf{Type}$. Now suppose we had a type $\mathit{type} : \textbf{Type}$ of small types and $\mathbb{N} : \mathit{type}$. Now we cannot declare $0 : \mathbb{N}$ because $\mathbb{N} : \mathit{type}$ is an object, and thus cannot type another term.

As the name says, the particularity of the $\lambda\Pi$-calculus when comparing with the λ-calculus is the addition of dependent types[2]. For instance, consider the successor function $S : \mathbb{N} \to \mathbb{N}$. For any element $n : \mathbb{N}$, the application Sn always gives a term that lives in $\mathbb{N}$. However, if we consider the equality type for natural numbers $= : \mathbb{N} \to \mathbb{N} \to \textbf{Type}$ (written infix) and we consider a function $\mathit{refl} : \Pi n : \mathbb{N}.n{=}n$ giving a proof of $n{=}n$ for every n, then for each element n the application $\mathit{refl}\ n$ lives on a different type. Indeed, $\mathit{refl}\ 0$ is of type $0 = 0$ but not of type $1 = 1$, because it is not a proof of $1 = 1$. This is because the term refl has a type which is dependent: its codomain depends on the argument given.

To resume things, we can give the following characterization of the hierarchy of the types in the $\lambda\Pi$-calculus.



The $\lambda\Pi$-calculus as a logical framework

The $\lambda\Pi$-calculus is well known for being in propositions as types correspondence with intuitionistic predicate logic[3]. Through the Curry-Howard correspondence, a proposition P is seen as a type P and the proofs of P are the inhabitants of this type. For instance, if P is a proposition, we represent the trivial proof of $P \Rightarrow P$ by $\lambda x : P.x$.

However, there is also a different approach to expressing predicate logic in the $\lambda\Pi$-calculus, known as *judgment as types*. Here, instead of representing a proposition directly as an inhabitant of **Type**, we declare a new type $\mathit{Prop} : \mathbf{Type}$ of propositions and a function $\mathit{Proof} : \mathit{Prop} \to \mathbf{Type}$ associating to each proposition a type of its proofs. We then add other constants to express each connective.[4] For instance, to add implication we add the constant $\Rightarrow : \mathit{Prop} \to \mathit{Prop} \to \mathit{Prop}$ (written infix) and the constants

$$\Rightarrow_{in} : \Pi a\ b : \mathit{Prop}.(\mathit{Proof}\ a \to \mathit{Proof}\ b) \to \mathit{Proof}\ (a \Rightarrow b)$$
$$\Rightarrow_{el} : \Pi a\ b : \mathit{Prop}.\mathit{Proof}\ a \to \mathit{Proof}\ (a \Rightarrow b) \to \mathit{Proof}\ b\,.$$

Therefore, whereas in the proposition as types approach we have that $\lambda x : P.x$ is a proof of $P \Rightarrow P$, in the judgment as types this is expressed by the term $\Rightarrow_{in} P\ P\ (\lambda x : \mathit{Proof}\ P.x)$.

The judgment as types approach was the one originally proposed by [8] to be used with the $\lambda\Pi$-calculus, when seeing it as a logical framework. Whereas the propositions as types puts this system in a "canonical" correspondence with predicate logic, when using the judgment as types approach we are able to encode many other system that are not in any correspondence to the $\lambda\Pi$-calculus. Indeed, it turns out that by using this method we are capable of representing many other type systems with features that are orthogonal to those of the $\lambda\Pi$-calculus, such as System F. This can seem very surprising, as we are capable of expressing a system with polymorphism in a system without it.

If we take another look at the encoding of predicate logic, we can note an important point we did not yet discuss. A proof of $P \vdash P$ with a cut is represented through the Curry-Howard correspondence by $(\lambda x : P.x)\alpha_P$ — where α_P represents the proof of P in the context —, whereas through the judgment as types approach we get the term $\Rightarrow_{el} P\ P\ \alpha_P\ (\Rightarrow_{in} P\ P\ (\lambda x : \mathit{Proof}\ P.x))$. On the first case, we have $(\lambda x : P.x)\alpha_P \longrightarrow \alpha_P$ and thus the term reduces to the representation of the cut-free proof. However, by using the last approach we lose this computational behavior, as the term $\Rightarrow_{el} P\ P\ \alpha_P\ (\Rightarrow_{in} P\ P\ (\lambda x : \mathit{Proof}\ P.x))$ is stuck and cuts do not reduce anymore.

As pointed out by Assaf[2], encodings such as this one, which lack preservation of computation, fail to be sound for more higher-order systems, such as for the Calculus of Constructions, as it cannot simulate proof reduction, β-reduction or other forms of computation. Thus, even though we are capable of encoding systems such as predicate logic and System F in a sound and complete way, the rigidity of the computation on the $\lambda\Pi$-calculus prevents us from going further.

Enriching computation in the $\lambda\Pi$-calculus

In 2007, Dowek and Cousineau considered in [5] an extension of the $\lambda\Pi$-calculus in which the notion of computation can be extended by adding rewriting rules. The syntax and the typing rules are kept the same, however they consider a more general notion of equivalence than only $\equiv_\beta$. More precisely, given a set $\mathcal{R}$ of rewriting rules — pairs of the form $cM_1..M_k \longrightarrow N$, where c is a constant and $M_1, ..., M_k, N \in \Lambda_{\lambda\Pi}$ —, the relation $\equiv$ in the $\lambda\Pi/\mathcal{R}$-calculus is defined as the least equivalence relation containing $\equiv_\beta$ and the context and substitution closure of the rules in $\mathcal{R}$. Given a rewrite rule $cM_1..M_k \longrightarrow N$, we normally call c its head symbol, $M_1..M_k$ its patterns and N its body.

By addressing the poorness of computation in the $\lambda\Pi$-calculus and adding the possibility of extending rewriting, the $\lambda\Pi/\mathcal{R}$-calculus becomes capable of expressing much richer systems. In [5], Dowek and Cousineau showed that we can express any functional PTS in a sound and complete way, which was already not possible in the $\lambda\Pi$-calculus. Since then, researchers in Deducteam have built on top of this work and proposed encodings of much richer features in the $\lambda\Pi/\mathcal{R}$-calculus, such as inductive types[3], universe polymorphism[7], cumulativity[2][12], proof-irrelevance[9], etc.

Expressing logics in the $\lambda\Pi$-calculus modulo rewriting

Let's now take a second try at doing a judgment as types encoding of logic, but now using rewrite rules. In the $\lambda\Pi$-calculus, we declared a constant $\Rightarrow$ to represent implication and we had to declare constants $\Rightarrow_{in}$ and $\Rightarrow_{el}$ to represent the introduction and elimination rules for this connective. However, a much nicer approach is possible in the $\lambda\Pi/\mathcal{R}$-calculus: we can just declare a rewrite rule identifying proofs of $a \Rightarrow b$ with functions from $\mathit{Proof}\ a$ to $\mathit{Proof}\ b$.

$$\mathit{Proof}\ (a \Rightarrow b) \longrightarrow \mathit{Proof}\ a \to \mathit{Proof}\ b$$

Now we don't need to declare constants for the introduction and elimination of implication, because these can be simulated by abstraction and application. For instance, a proof of $P \Rightarrow P$ can be simply given by the term $\lambda x : \mathit{Proof}\ P.x$.

We also recover the computational behavior: the representation of the proof of $P \vdash P$ containing a cut is now given by $(\lambda x : \mathit{Proof}\ P.x)\alpha_P$. We thus have $(\lambda x : \mathit{Proof}\ P.x)\alpha_P \longrightarrow \alpha_P$ and proofs with cuts now reduce to cut-free proofs.

To have a better understanding of how such encodings work, let's have a full look at the representation of predicate logic. However, before starting, we first establish a convention on how we represent encodings. First, declarations in the $\lambda\Pi$-calculus modulo rewriting are either constants, which are added to the signature Σ, or rewrite rules, which are added to $\mathcal{R}$. Therefore, each declaration will be marked either by (c-decl), meaning that the constant c is added to the signature Σ, or by (c-red), meaning that a rewrite rule concerning the constant c is added to $\mathcal{R}$. We also enclose such declarations by two vertical black bars (as seen in the next page), to explicit that we are making a constant or rule declaration.

We already have declared

$$\begin{array}{l} \mathit{Prop} : \mathbf{Type} \qquad (\mathit{Prop}\text{-decl}) \\ \mathit{Proof} : \mathit{Prop} \to \mathbf{Type} \qquad (\mathit{Proof}\text{-decl}) \\ \Rightarrow : \mathit{Prop} \to \mathit{Prop} \to \mathit{Prop} \qquad (\Rightarrow\text{-decl}) \\ \mathit{Proof}\ (a \Rightarrow b) \longrightarrow \mathit{Proof}\ a \to \mathit{Proof}\ b \qquad (\Rightarrow\text{-red}) \end{array}$$

which encodes the implicational fragment. To add first order quantification, we need to add a constant to represent the domain of discourse. If we want however to represent many-sorted predicate logic, in which we can have many sorts of domains of discourse, we can declare a type $\mathit{Set} : \mathbf{Type}$ which represent the set of sorts[5]. Now we can add multiple constants of type Set to represent different sorts of the language. For this example we only declare $\iota : \mathit{Set}$, which defines an one-sorted fragment of predicate logic. Finally, just like we had to declare a constant $\mathit{Proof} : \mathit{Prop} \to \mathbf{Type}$ which gives to each proposition a type of its proofs, we also need to declare $\mathit{El} : \mathit{Set} \to \mathbf{Type}$, associating to each sort of the language a type of its elements. In this case, sometimes we say that ι is a code in Set for the type $\mathit{El}\ \iota$.

$$\begin{array}{l} \mathit{Set} : \mathbf{Type} \qquad (\mathit{Set}\text{-decl}) \\ \iota : \mathit{Set} \qquad (\iota\text{-decl}) \\ \mathit{El} : \mathit{Set} \to \mathbf{Type} \qquad (\mathit{El}\text{-decl}) \end{array}$$

Now, given a sort x, we can declare universal quantification as a function which takes a term of type $\mathit{El}\ x \to \mathit{Prop}$ to a term Prop. More formally, we declare

$$\forall : \Pi x : \mathit{Set}.(\mathit{El}\ x \to \mathit{Prop}) \to \mathit{Prop} \qquad (\forall\text{-decl})$$

which allows us to represent $\forall_\iota x.P$ by $\forall\ \iota\ (\lambda x : \mathit{El}\ \iota.P)$. Finally, to have the proper introduction, elimination and computational behavior we add the rule

$$\mathit{El}\ (\forall\ A\ P) \longrightarrow \Pi x : \mathit{El}\ A.\mathit{Proof}\ (P\ x) \qquad (\forall\text{-red})$$

saying that an element of $\forall\ A\ P$ is simply a function taking an element x of type A and returning a proof of $P\ x$. It can show that this set of constants and rewrite rules provides a sound and complete encoding of (minimal intuitionistic) predicate logic.

Actually, in [4] researchers from Deducteam proposed a theory containing the one just presented which is capable of expressing in a unified way many systems and logics in the $\lambda\Pi/\mathcal{R}$-calculus, such as (intuitionistic and classic) predicate logic, higher order logic, the Calculus of Constructions, etc. We can thus see that the expressivity of the $\lambda\Pi/\mathcal{R}$-calculus makes it a very good candidate to be used as a logical framework and universal proof checker.

Dedukti and Lambdapi: implementing the $\lambda\Pi$-calculus modulo rewriting

Of course, if we want to use the $\lambda\Pi/\mathcal{R}$-calculus as a practical logical framework, we need to have some real implementation of it. Dedukti and its newer brother Lambdapi are two implementations of this system, and are used in practice to represent and check proofs. Many translators to and from Dedukti have already been developed or are in development[12][11][2], and most notably the encyclopedia of formal proofs expressed in Dedukti *Logipedia*[6] is one of the main projects at Deducteam.

1.2 Coinduction

Induction is a technique widely used in mathematics, which allows to define and reason about finitely constructed objects, such as integers, lists and trees. The well-foundedness of these objects is key in order to have their induction principles. However, by dropping the well-foundedness condition we find new objects which, although less used, are actually very useful when doing mathematics. This new technique, called Coinduction, allows us to represent possibly infinite objects, such as infinite lists, infinite trees but also formal languages, non well-founded sets and Böhm trees. Because of its usefulness, coinduction is nowadays present in many proof assistants, such as Coq, Agda and PVS. In order to understand coinduction and its relation to induction, we present its basis in this subsection.

(Co)Inductive Types

Inductive types are well known by most proof assistant users. By defining a type A and a set of constructors for A (satisfying a certain set of constraints, so we have a nice metatheory), the elements of the inductive type A are defined as the smallest set of terms stable by these constructors[6]. For instance, we can declare the type $\mathit{List}\ \mathbb{N}$ of lists of natural numbers, with constructors $[] : \mathit{List}\ \mathbb{N}$ for the empty list and $(\_ :: \_) : \mathbb{N} \to \mathit{List}\ \mathbb{N} \to \mathit{List}\ \mathbb{N}$ for adding an element to a list. Then we declare the elements of $\mathit{List}\ \mathbb{N}$ as the smallest set of terms closed by these constructors, that is, the least fixed point of the function

$$\phi : X \mapsto \{[]\} \cup \{n :: x \mid x \in X, n : \mathbb{N}\},$$

which can be described by $\cup_i \phi^i(\emptyset)$.These are exactly the terms that can be constructed by finitely applying the type's constructors and are exactly the lists of natural numbers.

Like many objects in mathematics, inductive types have a form of dual, called coinductive types. By defining a type A and a set of constructors for A, the elements of the coinductive type A are defined as the largest set of terms stable by these constructors. If now we interpret the same set of constructors for $\mathit{List}\ \mathbb{N}$ coinductively, we get the coinductive type $\mathit{Stream}\ \mathbb{N}$. Its elements form the largest set of terms closed by these constructors, that is, the greatest fixed point of

$$\phi : X \mapsto \{[]\} \cup \{n :: x \mid x \in X, n : \mathbb{N}\}.$$

which can be described by $\cap_i \phi^i(\Lambda)$, where Λ is the set of all terms.

When we consider only finite terms, both sets of inductive and coinductive types collapse to the same one. However, if we allow for infinite terms, the set of elements of $\mathit{List}\ \mathbb{N}$ stays the same, but we now get new elements of the type $\mathit{Stream}\ \mathbb{N}$. For instance, the term $0 :: 0 :: 0\ldots$ is stable by $0 :: \_$ and thus it is an element of $\mathit{Stream}\ \mathbb{N}$. Therefore, in the rest of this subsection, we will consider Λ to represent the set of infinitary lambda terms.

There is a very important point about this duality. If we analyze the equation $\mathit{List}\ \mathbb{N} = \cup_i \phi^i(\emptyset)$, this says that to construct the elements of $\mathit{List}\ \mathbb{N}$ we first start with the empty set and at each step we construct a new set of terms by adding the empty list and by applying $n :: \_$ to previous terms. At the end we find exactly the terms which can be finitely (in at most i steps, for some i) built with these constructors.

On the other hand, the equation $\cap_i \phi^i(\Lambda)$ tells another story: we first start with all terms and at each step we build a new set by eliminating terms which are both different of the empty list and cannot be deconstructed as $n :: x$, for some term x in the previous set. At the end we find exactly the terms which can be destructed arbitrarily many times through the constructors.

The duality here is very clear: whereas elements of inductive types are built by constructing elements with constructors from scratch, elements of coinductive types are built by starting with everything and eliminating those which cannot be observed as constructors. Therefore, whereas induction is about building things, coinduction is about destructing them. We will also see on the next part that when looking at the induction and coinduction principles this gets inverted: whereas the induction principle allows us to destruct an element of an inductive type, the coinduction principle allows us to build an element into a coinductive type.

(Co)Recursion

In order to understand how inductive and coinductive types can be used, we need to look at their principles. Although it would be faster to just state them, it is much more intuitive to see how we can naturally recover them from a categorical semantics of induction and coinduction.

Consider the category **Set** of sets and the endofunctor $F : X \mapsto 1 + \mathbb{N} \times X$ with its obvious action on morphisms (note that F corresponds somewhat to the function ϕ seen previously). We can then build the category $\mathbf{Set}_F$ of F-algebras, whose objects are functions (e.g., morphisms in **Set**) of the form $\alpha_A : 1 + \mathbb{N} \times A \to A$ and morphisms in $\mathbf{Set}_F(\alpha_A, \alpha_B)$ are functions $f : A \to B$ making the following diagram commute.

$$\begin{array}{ccc} 1 + \mathbb{N} \times A & \xrightarrow{Ff} & 1 + \mathbb{N} \times B \\ \downarrow{\scriptstyle \alpha_A} & & \downarrow{\scriptstyle \alpha_B} \\ A & \xrightarrow{f} & B \end{array}$$

We can show that $\mathbf{Set}_F$ has the terminal object $\mathit{List}\ \mathbb{N}$, and with $\alpha_{\mathit{List}\ \mathbb{N}}$ defined by $* \mapsto []$ and $(n, l) \mapsto n :: l$. Then by initiality, for each $\alpha_A : 1 + \mathbb{N} \times A \to A$, there is a unique function $\mathit{rec}\ \alpha_A$ making the following diagram commute.

$$\begin{array}{ccc} 1 + \mathbb{N} \times \mathit{List}\ \mathbb{N} & \xrightarrow{F(\mathit{rec}\ \alpha_A))} & 1 + \mathbb{N} \times A \\ \downarrow{\scriptstyle \alpha_{\mathit{List}\ \mathbb{N}}} & & \downarrow{\scriptstyle \alpha_A} \\ \mathit{List}\ \mathbb{N} & \xrightarrow{\mathit{rec}\ \alpha_A} & A \end{array}$$

We see the induction principle naturally arise.

Induction Principle: Each $\alpha_A : 1 + \mathbb{N} \times A \to A$ defines a unique function $\mathit{rec}\ \alpha_A : \mathit{List}\ \mathbb{N} \to A$;

A theorem by Lambek ensures that $\alpha_{\mathit{List}\ \mathbb{N}}$ is actually an isomorphism, so we can actually inverse it and find an explicit expression for $\mathit{rec}\ \alpha_A$ as $\alpha_A \circ F(\mathit{rec}\ \alpha_A) \circ \alpha^{-1}_{\mathit{List}\mathbb{N}}$. By separating the cases when the list is empty or not, we find

$$\begin{aligned} (\mathit{rec}\ \alpha_A)\ [] &= \alpha_A\ (*) \\ (\mathit{rec}\ \alpha_A)\ (n :: l) &= \alpha_A\ (n, (\mathit{rec}\ \alpha_A)\ l)\,. \end{aligned}$$

This is quite revealing: we see that defining the function α_A is actually pattern matching with primitive recursion. For instance, if we take $A = \mathbb{N}$ and $\alpha_{\mathbb{N}}$ defined by $* \mapsto 0$ and $(n, \mathit{len}) \mapsto \mathit{len} + 1$, we get

$$\begin{aligned} (\mathit{rec}\ \alpha_{\mathbb{N}})\ [] &= 0 \\ (\mathit{rec}\ \alpha_{\mathbb{N}})\ (n :: l) &= ((\mathit{rec}\ \alpha_{\mathbb{N}})\ l) + 1\,, \end{aligned}$$

the definition of the function *length* on lists.

Likewise, we can build the category $\mathbf{Set}^F$ of F-coalgebras, whose objects are functions of the form $\beta_A : A \to 1 + \mathbb{N} \times A$ and morphisms in $\mathbf{Set}^F(\beta_A, \beta_B)$ are functions $f : A \to B$ making the following diagram commute.

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ \downarrow{\scriptstyle \beta_A} & & \downarrow{\scriptstyle \beta_B} \\ 1 + \mathbb{N} \times A & \xrightarrow{Ff} & 1 + \mathbb{N} \times B \end{array}$$

We can then show that $\mathbf{Set}^F$ has the terminal object $\mathit{Stream}\ \mathbb{N}$, with $\beta_{\mathit{Stream}\ \mathbb{N}}$ defined by $[] \mapsto *$ and $n :: l \mapsto (n, l)$. Then by finality, for each $\beta_A : A \to 1 + \mathbb{N} \times A$, there is a unique function $\mathit{corec}\ \alpha_A$ making the following diagram commute, thus yielding the coinduction principle.

$$\begin{array}{ccc} A & \xrightarrow{\mathit{corec}\ \beta_A} & \mathit{Stream}\ \mathbb{N} \\ \downarrow{\scriptstyle \beta_A} & & \downarrow{\scriptstyle \beta_{\mathit{Stream}\ \mathbb{N}}} \\ 1 + \mathbb{N} \times A & \xrightarrow{F(\mathit{corec}\ \beta_A)} & 1 + \mathbb{N} \times (\mathit{Stream}\ \mathbb{N}) \end{array}$$

Coinduction Principle: Each $\beta_A : A \to 1 + \mathbb{N} \times A$ defines a unique function $\mathit{corec}\ \beta_A : A \to \mathit{Stream}\ \mathbb{N}$.

We first remark a very important point. Whereas functions defined by induction eliminates from an inductive type to an arbitrary type, coinductive functions do the opposite, eliminating from an arbitrary type to a coinductive one. Therefore, a function that eliminates from a coinductive type to an arbitrary one cannot be coinductive, just like a function that builds an inductive type from an arbitrary one cannot be inductive neither.

A second theorem by Lambek ensures also that $\beta_{Stream\ \mathbb{N}}$ is an isomorphism, allowing us to write $corec\ \beta_A = \beta^{-1}_{Stream\ \mathbb{N}} \circ F(corec\ \beta_A) \circ \beta_A$. By doing a case analysis on the value of $\beta_A\ x$, we can then find the following expression for $corec\ \beta_A$.

$$(corec\ \beta_A)\ x = \begin{cases} [] & \text{if } \beta_A\ x = * \\ n :: ((corec\ \beta_A)(y)) & \text{if } \beta_A\ x = (n, y) \end{cases}$$

If we take, for instance, $A = \mathbb{N}$ and $\beta_A : n \mapsto (n, n+1)$ we get the equation

$$(corec\ \beta_{\mathbb{N}})\ n = n :: ((corec\ \beta_{\mathbb{N}})\ (n+1)),$$

which maps each integer n to the stream $n, n+1, n+2, \ldots$ — we will call this function *natStream*, as we will use it as a recurring example.

We could take this clause as a definition of *natStream*, however as it refers to itself in a non well-founded way, it's clear that it can cause non-termination issues. Even though coinductive types are non well-founded by definition, in practice when dealing with real proof assistants and programming languages, we need to have a finitary way to deal with these objects, as terms are always finite in these situations. We will discuss in one of the other parts how this problem can be handled.

The derived equations for recursion and corecursion also allows us to see how they are characterized syntactically. When defining $rec\ \alpha_c$ by recursion, the definition $(rec\ \alpha_c)\ (n :: l)$ can only be a function of n and $(rec\ \alpha_c)\ l$. This means that any recursive call can only be made on the recursive argument of the constructor — l in this case. We shall also remark that most proof assistants, like Coq, allow for a more general form of recursion, which allow for recursive calls to be performed in any argument structurally smaller.

On the other hand, when defining $corec\ \beta_c$ by corecursion, we can send its value either to the empty stream, or to a new instance of the constructor _ :: _, in which the second argument must be the result of a corecursive call to $corec\ \beta_c$. Therefore, the body of the corecursive function must be directly equal to an instance of the constructor, and corecursive calls must be given as direct argument to constructors, as it is the case in $(corec\ \beta_{\mathbb{N}})\ n = n :: ((corec\ \beta_{\mathbb{N}})\ (n+1))$. When discussing the implementation of coinduction in Agda we will see in more depth these criteria.

1.3 The Agda proof assistant

Agda is a dependently-typed programming language developed in Sweden, mainly used as a proof assistant. Its type system extends Martin-Löf Type Theory with many features, such as (co) inductive types, universe polymorphism, sized types, etc. On the following, we take a look at some of its main characteristics.

Universes and type system

Just like in Martin-Löf Type Theory, Agda features an infinite hierarchy of universes $Set_0 : Set_1 : Set_2 \ldots$, such that any type must be typed by a universe — note that universes themselves satisfy this criteria, as each universe Set_i is typed by the universe Set_{i+1}. We call the integers indexing the universes *universe levels*. We also might refer to universes as sorts. Most of the time, we write *Set* when referring to Set_0. We also note that, unlike Coq and Martin-Löf Type Theory, Agda does not feature comulativity by default, which is the ability of raising a type A living in a universe Set_i to the universe Set_j when $i < j$.[7]

Given two types $A : Set_i, B : Set_j$, with B possibly containing a free variable x of type A, we can form the dependent product type $(x : A) \to B$ (the Agda notation for $\Pi x : A.B$) using the following rule, where $\sqcup$ calculates the maximum between two levels.

$$\frac{\Sigma; \Gamma \vdash A : Set_i \qquad \Sigma; \Gamma, x : A \vdash B : Set_j}{\Sigma; \Gamma \vdash (x : A) \to B : Set_{i \sqcup j}}$$

Finally, one of the main particularities of Agda, when compared with most proof assistants such as Coq, HOL, Lean, etc, is that Agda's type system does not separate propositions from "normal" types. For instance, whereas in Coq the type $\mathbb{N}$ of natural numbers lives in *Set* and the proposition true $\top$ lives in *Prop*, in Agda both of them live in *Set*[8]. An important consequence is that, whereas in Coq the lambda term $\lambda x : A.x$ can either be the identity function for A (if $A : Set$) or a

proof of $A \Rightarrow A$ (if $A : Prop$), in Agda the term $\lambda x : A.x$ is both the identity and a proof of $A \Rightarrow A$ at the same time. It all depends if we prefer to interpret A as a normal type or a type representing a proposition. Therefore, Agda implements a radical version of Curry-Howard, making absolutely no difference between types ans propositions.

Inductive types, recursive functions and records

Like most proof assistants, Agda features inductive types. Though its presentation is mostly standard, its main peculiarity is that elimination principles are not defined explicitly. Rather, elimination from an inductive type is done by defining a recursive function with clauses (in a Haskell-like manner) which then needs to pass the language's termination and totality checkers.

For instance, we can declare the inductive type of natural numbers by (Agda code bellow)

```
data Nat : Set where
  zero : Nat
  succ : Nat → Nat
```

and then define the sum of two natural numbers by induction on the first element using the following definition.

```
_+_ : Nat → Nat → Nat
zero + x = x
(succ y) + x = succ (y + x)
```

Another particularity of Agda is that it also feature records, which are basically inductive types with one constructor and special treatment (as we will see on the next parts). For instance, we can define the type dependent pairs with the following record definition.

```
record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    fst : A
    snd : B fst
```

We note that the fields are also called projections, as they can be used as eliminators. For instance, the application fst (a, b) for $(a, b) : \Sigma\ A\ B$ reduces to the value of a.

Universe polymorphism

As already said, Agda extends Martin-Löf Type Theory in a number of ways. One of these new features is the addition of *universe polymorphism*, which allows for building terms which can live in multiples universes. In order to understand what this means, suppose we want to define an inductive type for lists, associating to each type A in Set another type in Set of lists of A.

```
data List (A : Set) : Set where
  cons : A → List A → List A
  nil : List A
```

However, Set in Agda is just an alias for Set_0, as we have an infinite hierarchy $Set_0 : Set_1 : \ldots$ of sorts. Thus if we have a type B which lives in Set_1 we need to declare another inductive type $List_1 : Set_1 \to Set_1$ to build lists of elements in B, and so on if we have a type C living in Set_2. Universe polymorphism allows us to build the inductive type

```
data List (i : Level) (A : Set i) : Set i where
  cons : A → List i A → List i A
  nil : List i A
```

which can then be instantiated at each level, avoiding the declaration of an infinite number of versions of $List$. Universe polymorphism also allows us to build functions that can deal with types in multiple universes, such as the universe polymorphic identity function below.

```
id-poly : (i : Level) → (A : Set i) → A → A
id-poly i A x = x
```

In order to give a type to terms like $(i : Level) \to (A : Set\ i) \to A \to A$, Agda introduces a sort $Set\omega$ of universe polymorphic type. Using this sort, we can build types which feature *prenex* universe polymorphism, in which the level quantification occurs in the outer part of the term. This is what ensures us that the polymorphic definitions *List* and *id-poly* are indeed part of Agda's type theory.

However, Agda has also recently added a second sort hierarchy $Set\omega_0 : Set\omega_1 : ...$, which then also allows for *non-prenex* universe polymorphism, in which level quantification can appear anywhere in the term. This allows for instance to build the following function, which applies a universe polymorphic function f to a universe polymorphic value x to get another universe polymorphic value. The omega hierarchy also fixes a missing symmetry, as before the sort $Set\omega$ was the only sort not having a type. By adding the omega hierarchy, each $Set\omega_i$ is now typed by $Set\omega_{i+1}$.

```
app-poly : ((i : Level) → Set i → Set i) → ((i : Level) → Set i) → (i : Level) → Set i
app-poly f x i = (f i) (x i)
```

η-equivalence

Like some proof assistants (and unlike the $\lambda\Pi$-calculus modulo rewriting), the Agda conversion system features η-equivalence, a kind of dual to β-equivalence. Whereas β-reduction explains how to reduce an eliminator (application) applied to a constructor (abstraction)

$$(\lambda x : A.M)N \longrightarrow_\beta M(N/x)\,,$$

η-expansion explains how to expand to a constructor applied to an eliminator[9]

$$f \longrightarrow_\eta (\lambda x : A.f\ x) \qquad \text{with } f : \Pi x : A.B\,.$$

Of course, starting from a term $f : A \to B$ we could keep η-expanding to infinity, as this is a non-terminating process. However, by expanding $\lambda x : A.f\ x$ we would create a β-redex, thus we don't do this step and we say that $\lambda x : A.f\ x$ is in *eta-long form*.

Note that a major difference between β-reduction and η-expansion is that whereas β-reduction can be defined in an untyped setting (as in the untyped λ-calculus), η-expansion needs to inspect the type of the term in order to know if it is η-expandable. Therefore, we say that η-expansion is a *type-directed* computation rule.

In the Agda system, this rule is also defined for most record types[10]. For instance, if we consider the previously defined record of dependent pairs, if *pair* is an element of $\Sigma\ \mathsf{Nat}\ (\lambda\_.\mathsf{Nat})$ (the type of pairs of natural numbers), then we have

$$pair \equiv (fst\ pair, snd\ pair)\,.$$

We then say that the term $(fst\ pair, snd\ pair)$ is in eta-long form and we do not expand it anymore.

1.4 Agda2Dedukti: a practical translator into the $\lambda\Pi$-calculus modulo rewriting

The problem of encoding the logic of Agda in the $\lambda\Pi$-calculus modulo rewriting was first treated by Guillaume Genestier[7], who started the development of Agda2Dedukti, a prototype translator. In order to do so, he used techniques already known in the literature[x] to represent universes and inductive types, but he most notably proposed encodings of prenex universe polymorphism and eta conversion, two features whose representation in the $\lambda\Pi$-calculus modulo rewriting was not known. We present a quick review of how the encoding works.

Before starting, we adapt our convention on how to represent Dedukti encodings and we introduce the color green to represent constants alongside blue. Now we declare in blue constants which encode the underling type theory of a system, so for instance as the hierarchy $Set_0 : Set_1 : ...$ is primitive in the Agda type system and cannot be removed, it will be represented by constants in blue. On the other hand, we use green to declare constants which correspond to definitions in an Agda file and which are not primitive in the system, but added by the user, such as the definition

of natural numbers $\mathbb{N}$ and the function plus $+$. Variables and symbols which are primitive to the $\lambda\Pi$-calculus modulo theory, such as x, A, M, α, λ, Π, $\to$, **Type** are still represented in black. We remind that, even thought the colors blue and green are also used in the Agda, their meaning is completely different, and thus it is important to not mistake Agda code with Dedukti declarations. Finally, when declaring constants that play a role of function, we allow ourselves to write them in infix notation or to write some arguments as subscripts, when this simplifies the notation (see cases of $\sqcup$ and $\rightsquigarrow$ below).

Universes and type system

To represent the infinite hierarchy of universes $Set_0 : Set_1 : Set_2 : \ldots$, we first declare a type of sorts and a function that associates to each sort the type of its terms. Therefore, each Agda type A that lives in a sort or universe α will be represented as a term in the type $U\ \alpha$.

$$Sort : \mathbf{Type} \qquad (Sort\text{-decl})$$
$$U : Sort \to \mathbf{Type} \qquad (U\text{-decl})$$

Then we define a type L to represent universe levels (it would be more appropriated to call it $\mathbb{N}$, but we reserve this name to the integers as represented in the theory). Finally, we can define the sorts Set_i by defining a function associating to each level a sort.

$$L : \mathbf{Type} \qquad (L\text{-decl})$$
$$z : L \qquad (z\text{-decl})$$
$$s : L \to L \qquad (s\text{-decl})$$
$$set : L \to Sort \qquad (set\text{-decl})$$

We can already represent some basic types, like is the case with integers with $\mathbb{N} : U\ (Set\ z)$. However, to represent the elements that live in $\mathbb{N}$ we need another function, taking a sort α and a type A in $U\ \alpha$ and returning the type of elements of A.

$$El : \Pi\alpha : Sort.U\ \alpha \to \mathbf{Type} \ \text{(written as } El_\alpha) \qquad (El\text{-decl})$$

Now we can declare 0 as an element of $El_{set\ z}\ \mathbb{N}$. In Agda, we also have $Set_i : Set_{i+1}$ for each i, therefore we need to represent this on the encoding. To do so, we first define a function mapping each sort into its successor sort. Then we can declare a function mapping each sort α on the corresponding object that lives in $\square\ \alpha$. To properly identify the object $\square\ \alpha$ with the sort α we add a rewrite rule identifying their types.

$$\square : Sort \to Sort \qquad (\square\text{-decl})$$
$$\square\ (set\ i) \longrightarrow set\ (s\ i) \qquad (\square\text{-red})$$
$$\diamond : \Pi\alpha : Sort.U\ (\square\ \alpha) \qquad (\diamond\text{-decl})$$
$$El\ \_\ (\diamond\ \alpha) \longrightarrow U\ \alpha \qquad (\diamond\text{-red})$$

The joker on the last rule represents a non used variable. As this is constraint by typing as being equal to $\square\ \alpha$, we don't need to specify it. Finally it is only left to represent Agda dependent products, such as $(n : \mathbb{N}) \to n = n$. To do this, we first declare a constant calculating the max between two sorts. This function uses an auxiliary max function which operates on levels.

$$\sqcup : L \to L \to L \ \text{(written infix)} \qquad (\sqcup\text{-decl})$$
$$(s\ x) \sqcup (s\ y) \longrightarrow s\ (x \sqcup y) \qquad (\sqcup\text{-red})$$
$$z \sqcup x \longrightarrow x \qquad (\sqcup\text{-red})$$
$$x \sqcup z \longrightarrow x \qquad (\sqcup\text{-red})$$
$$\vee : Sort \to Sort \to Sort \ \text{(written infix)} \qquad (\vee\text{-decl})$$
$$(set\ i) \vee (set\ j) \longrightarrow set\ (i \sqcup j) \qquad (\vee\text{-red})$$

Now we can define the constant representing products, which takes two sorts α, β, a type A in α and a function mapping each element of A to a type in β.

$$\rightsquigarrow : \Pi(\alpha\ \beta : \mathit{Sort})(A : \mathit{U}\ \alpha).(\mathit{El}_\alpha\ A \to \mathit{U}\ \beta) \to \mathit{U}\ (\alpha \vee \beta) \text{ (written infix as } {}_\alpha\!\rightsquigarrow_\beta) \quad (\rightsquigarrow\text{-decl})$$
$$\mathit{El}_{\_}\ (A\ {}_\alpha\!\rightsquigarrow_\beta\ B) \longrightarrow \Pi x : \mathit{El}_\alpha\ A.\mathit{El}_\beta\ (B\ x) \quad (\rightsquigarrow\text{-red})$$

For instance, if we want to represent $\mathit{refl} : (n : \mathbb{N}) \to n = n$ we can declare the constant *refl* as living in

$$\mathit{El}_{\mathit{set}\ z}\ (\mathbb{N}\ {}_{(\mathit{set}\ z)}\!\rightsquigarrow_{(\mathit{set}\ z)}\ (\lambda n : \mathit{El}_{\mathit{set}\ z}\ \mathbb{N}.n{=}n)),$$

as the latter reduces to $\Pi n : \mathit{El}_{\mathit{set}\ z}\ \mathbb{N}.\mathit{El}_{\mathit{set}\ z}\ (n{=}n)$. In this case we could ask ourselves why not define the type of *refl* directly as being $\Pi n : \mathit{El}_{\mathit{set}\ z}\ \mathbb{N}.\mathit{El}_{\mathit{set}\ z}\ (n{=}n)$. However, for a type to be in the image of the translation, it must be convertible to a type of the form $\mathit{El}_\alpha\ A$, for some α, A. Therefore, by writing it in this more complicated way, we explicit the fact that *refl* lives in the translation of an Agda type, and not just in some arbitrary type in Dedukti.

Nevertheless, as this encoding makes the presentation of the work a lot heavier, we will make a choice to simplify the notation on the next parts. Explicitly, we will write a product type such as $\mathit{El}_{\alpha\vee\beta}\ (A\ {}_\alpha\!\rightsquigarrow_\beta\ B)$ directly as its normal form, and we will write $\mathit{El}\ A$ to represent $\mathit{El}_{\mathit{set}\ z}\ A$. Therefore, the type

$$\mathit{El}_{\mathit{set}\ z}\ (\mathbb{N}\ {}_{(\mathit{set}\ z)}\!\rightsquigarrow_{(\mathit{set}\ z)}\ (\lambda n : \mathit{El}_{\mathit{set}\ z}\ \mathbb{N}.n{=}n))$$

of *refl* gets represented as

$$\Pi n : \mathit{El}\ \mathbb{N}.\mathit{El}\ (n{=}n).$$

However, this simplification is only for presentation purposes, and is not used in practice in the real translation.

Inductive types and recursive functions

The representation of Agda's inductive types and recursive functions can be done in a very simple way. For each inductive definition, such as the one of natural numbers we declare a constant representing the type and one constant to represent each constructor.

$$\mathbb{N} : \mathit{U}\ (\mathit{set}\ z) \quad (\mathbb{N}\text{-decl})$$
$$\mathit{zero} : \mathit{El}\ \mathbb{N} \quad (\mathit{zero}\text{-decl})$$
$$\mathit{succ} : \mathit{El}\ \mathbb{N} \to \mathit{El}\ \mathbb{N} \quad (\mathit{succ}\text{-decl})$$

In order to translate a recursive function, such as the sum, we first declare a constant defining the function and we add a rewrite rule for each clause. Note that, as these rules are only fired when the left term corresponds to an instance of the constructor, when translating a terminating Agda function, we obtain automatically a terminating set of rewrite rules.

$$+ : \mathit{El}\ \mathbb{N} \to \mathit{El}\ \mathbb{N} \to \mathit{El}\ \mathbb{N} \quad (+\text{-decl})$$
$$\mathit{zero}{+}y \longrightarrow y \quad (+\text{-red1})$$
$$(\mathit{succ}\ x){+}y \longrightarrow \mathit{succ}\ (x{+}y) \quad (+\text{-red2})$$

On the definition of the translated function $+$, as there was no overlap between clauses, the translation was immediate. However, consider the following definition with overlapping clauses.

```
test1 : Nat → Bool
test1 (succ zero) = true
test1 _ = false
```

If we were to translate this definition naively we would have the following rewrite rules.

$$\mathit{test1}\ (\mathit{succ}\ \mathit{zero}) \longrightarrow \mathit{true}$$
$$\mathit{test1}\ \_ \longrightarrow \mathit{false}$$

However, because rewrite rules have no priority order, in this system we would have both $test1\ (succ\ zero) \longrightarrow true$ and $test1\ (succ\ zero) \longrightarrow false$. In order to solve this problem we must make sure that the reduction respects the semantics of Agda, meaning that we can only move to the next clause when we are sure that there is no possible match with the current one.

Fortunately, Agda's internal coverage check, which is used to check totality of clauses, also compiles them into a non-overlapping set of clauses. For instance, by translating the clauses produced by the coverage check, we obtain the following rewrite rules for the function *test*1. This allows the translation of functions to correctly reflect the semantics of their definitions in Agda.

$$\begin{aligned} test1\ zero &\longrightarrow false \\ test1\ (succ\ zero) &\longrightarrow true \\ test1\ (succ\ (succ\ \_)) &\longrightarrow false \end{aligned}$$

Universe polymorphism

Consider once again the universe polymorphic type of lists.

```
data List (i : Level) (A : Set i) : Set i where
  cons : A → List i A → List i A
  nil : List i A
```

In order to represent this inductive type in Dedukti we would have first to declare the following constant to represent this type.

$$List : \Pi i : L.U\ (set\ i) \to U\ (set\ i) \qquad (List\text{-decl})$$

However, remember that in order for this term to live in our representation of Agda, we need to be able to declare it as living in a type of the form $El_\alpha\ A$, such that $El_\alpha\ A$ reduces to the expected type $\Pi i : L.U\ (set\ i) \to U\ (set\ i)$. In our current encoding this would not be possible, as we are not able yet to represent universe polymorphism.

In order to add this feature to our representation we first add a sort $set\omega$ to type universe polymorphic types. Next, we need to add a function taking types depending on a level and producing a universe polymorphic type in $set\omega$. To do this, we add the function $\forall$ taking a sort depending on a level $(\alpha : L \to Sort)$ and a type in $U\ (\alpha\ i)$ depending on a level i $(A : \Pi i : L.U\ (\alpha\ i))$, and producing a universe polymorphic type. Finally, we declare a rule asserting that the elements of $\forall\ \alpha\ A$ are indeed functions giving for each i a type in $El_{(\alpha\ i)}\ (A\ i)$.

$$\begin{aligned} &set\omega : Sort && (set\omega\text{-decl}) \\ &\forall : \Pi\alpha : L \to Sort.(\Pi i : L.U\ (\alpha\ i)) \to U\ set\omega && (\forall\text{-decl}) \\ &El\_\ (\forall\ \alpha\ A) \longrightarrow \Pi i : L.El_{(\alpha\ i)}\ (A\ i) && (\forall\text{-red}) \end{aligned}$$

Using this new encoding, we can now declare the constant *List* as having a type of the form $El_\alpha\ A$.

$$List : El_{set\omega}\ (\forall\ (\lambda i : L.set\ (s\ i))\ (\lambda i : L.(\diamond\ (set\ i))\ _{set\ (s\ i)}\!\rightsquigarrow_{set\ (s\ i)}\ (\lambda\_, \diamond\ (set\ i))))$$

We can also verify that its assigned type reduces to the expected one, as we have the reduction

$$El_{set\omega}\ (\forall\ (\lambda i : L.set\ (s\ i))\ (\lambda i : L.(\diamond\ (set\ i))\ _{set\ (s\ i)}\!\rightsquigarrow_{set\ (s\ i)}\ (\lambda\_, \diamond\ (set\ i)))) \longrightarrow \Pi i : L.U\ (set\ i) \to U\ (set\ i)\,.$$

An additional aspect of universe polymorphism which is much harder to represent is level conversion. In Agda, level conversion contains many identities which are semantically valid, such as $i \sqcup j \equiv j \sqcup i$, $(i \sqcup j) \sqcup k \equiv i \sqcup (j \sqcup k)$, $i \sqcup i \equiv i$, etc, and which are needed to be used when checking proofs and definitions. For instance, if we have a function $maxSet = \lambda ij : Level.Set_{i \sqcup j}$ taking two levels and yielding the highest instance of *Set*, then in Agdawe have the conversions $maxSet\ i\ j \equiv maxSet\ j\ i$ and $maxSet\ i\ i \equiv Set_i$. Even thought we also have this in Dedukti when we replace i and j by closed terms, for this to be true with variables we would have to have in Dedukti the conversions $i \sqcup j \equiv j \sqcup i$ and $i \sqcup i \equiv i$,

which is not true for the encoding we have looked at until now. In order to take this into account, an extension of the representation of levels was proposed by Genestier, which uses AC conversion and matching. We will not enter into its details here, as this will not be important for us, but we refer to [7] for more details.

Eta-conversion

As we have already seen, computation in Agda features eta-expansion, a rewrite rule which, different from most, is type-directed. This characteristic makes eta-expansion impossible to be directly expressed with our notion of rewrite rule. Indeed, in the $\lambda\Pi$-calculus modulo rewriting, a rewrite rule is a pair $l \longrightarrow r$ in which l is of the form $cl_1 \ldots l_k$. We call this kind of rewriting untyped because matching is done purely syntactically, and we cannot inspect the types of the terms in order to know if a rewrite rule is applicable. In contrast, η-expansion is defined by

$$f \longrightarrow \lambda x : A.f\ x \quad \text{if } f : \Pi x : A.B$$

and therefore we need to know the type of f in order to know if we can apply the rule. Moreover, this is not the only problem, as the applicability of this rule is also sensitive to the position on the term. Indeed, even if the term f appears in $\lambda x : A.f\ x$ with a type $\Pi x : A.B$, we cannot further expand it, as this would lead to non-termination issues.

Therefore, in order to solve this problem, we must both simulate a rule which is typed and prevent its non-termination loops. A possible solution is to introduce a symbol η allowing to annotate terms with their types.

$$\eta : \Pi(\alpha : \mathit{Sort})(A : U\ \alpha).A \to A \ \text{(written as } \eta_\alpha^A) \qquad (\eta\text{-decl})$$

Using this symbol, we can properly match on a term's type in order to define eta-expansion. For instance, if f has a product type $A\ {}_\alpha\!\rightsquigarrow_\beta\ B$, then it can be eta-expanded using the following rule.

$$\eta_{\_}^{A\ \alpha\rightsquigarrow_\beta\ B}\ f \longrightarrow \lambda x : El_\alpha\ A.\eta_\beta^{B\ x}\ (f\ x) \qquad (\eta\rightsquigarrow\text{-red})$$

Moreover, note that after the reduction, the application $f\ x$ gets annotated by the η symbol, and the outer abstraction is no more annotated. Therefore, we cannot apply the rule directly to f, and unless if $f\ x$ has also a product type, the rewrite rule cannot be reapplied, and thus we do not run into non-termination. We can also add rules to express the eta-expansion of records. For instance, the following rule adds eta-expansion to our translation of the record type of dependent pairs.

$$\eta_{\_}^{\Sigma\ A\ B}\ M \longrightarrow \mathit{pair}\ (\eta_{\mathit{set}\ z}^{A}\ (\mathit{fst}\ A\ B\ M))\ (\eta_{\mathit{set}\ z}^{B\ (\mathit{fst}\ A\ B\ M)}\ (\mathit{snd}\ A\ B\ M)) \qquad (\eta\Sigma\text{-red})$$

This presentation gives the general intuition behind this representation. Nevertheless, in order to define precisely how the translation of η-expansion fully works, we would have to deal with many nuances and technical details, which we thus prefer to omit here. We refer to [7] for more details.

2 Representing Coinduction in the $\lambda\Pi$-calculus modulo rewriting

As seen on the subsection about coinduction, the interesting point of this technique is being able to reason about objects which are possibly infinite. Obviously, this raises a problem when trying to implement coinduction in proof assistants, as terms must always be finite. This question is normally addressed by resorting to lazy representations of elements of coinductive types. This means that the infinite terms are never represented in their entirety, but can be developed an arbitrary amount of times, when required by the user.

In the $\lambda\Pi$-calculus modulo rewriting, it is clear that trying to define the function *natStream* naively by

$$\mathit{natStream}\ n \longrightarrow n :: (\mathit{natStream}\ (\mathit{succ}\ n))$$

would not work, as rewriting rules can be fired without any checks if the computation is really necessary. Therefore, we need to find a smarter way to control rewriting in this setting. In this section we will explore how this is handled in the case of Agda and we will see how such ideas can be reused in our setting, allowing to represent coinduction and to translate Agda proofs.

2.1 Coinduction in Agda

Coinduction in Agda features two presentations: musical coinduction, which is the old way of using coinduction in Agda, and copattern matching coinduction, which is nowadays considered the standard. Both presentations try to solve the non-termination problem presented when introducing coinduction.

Musical Coinduction

Musical coinduction addresses this problem by introducing the following control operators, which control explicitly the evaluation of the corecursive calls. The constant ∞ associates to each type A the type $\infty\ A$ of halted computations, whereas the operators $\sharp$ and $\flat$, also known as thunk and force, allow respectively to halt a computation and to resume it.

```
∞ : (A : Set) → Set
♯ : {A : Set} → A → ∞ A
♭ : {A : Set} → ∞ A → A
```

To define the coinductive type of streams using musical coinduction we use the following declaration.

```
data Stream (A : Set) : Set where
  _::_ : (x : A) (xs : ∞ (Stream A)) → Stream A
  [] : Stream A
```

The main particularity here is that the corecursive argument of the constructor _ :: _ now takes a halted computation of type *Stream A*. Therefore, to define the curecursive function *natStream* we need to use the $\sharp$ operator to halt the computation of *natStream* $(n + 1)$. As terms do not reduce under the $\sharp$, this definition is terminating.

```
natStream : Nat → Stream Nat
natStream n = n :: ♯ (natStream (succ n))
```

A halted calculation can be resumed by the symbol $\flat$, as expressed by the identity $\flat\ (\sharp\ x) = x$. Using this symbol, we can define for instance a function, which given a position n takes the n-th element of the stream, if it exists. In order to do this, we first need to introduce the type constructor *Maybe*, used when defining partial functions. Its constructors tell us that a value of type *Maybe A* is either a value of A or nothing.

```
data Maybe (A : Set) : Set where
  just : A → Maybe A
  nothing : Maybe A
```

We can now proceed to the definition of the function n-th. At each step, if the position we are looking for is zero we simply return the head, otherwise we resume the computation of the tail (using the $\flat$ operator) and we do a recursive call on it. If at any point we reach the end of the stream (as streams here are only possibly infinite, and not always), we return nothing to signal that the searched element does not exist. We note that, in opposition to *natStream*, which is defined by corecursion, *nth* is defined by recursion on its first argument.

```
nth : Nat → Stream Nat → Maybe Nat
nth zero (hd :: tl) = just hd
nth (succ n) (hd :: tl) = nth n (♭ tl)
nth _ [] = nothing
```

Copattern matching coinduction

Even though musical coinduction solves the problem of non-termination, it is clear that it is not very intuitive to use, as the user needs to control explicitly how calculations are halted and resumed, which requires some experience. Abel *et al* introduced in [1] a different presentation of coinduction through copattern matching. The main idea here is to restrict ourselves to one constructor coinductive types expressed as records and then define each element not through constructors but rather through their eliminators, or projections.

For instance, if we consider the coinductive type of streams defined only by the constructor $\_ :: \_ : A \to \mathit{Stream}\ A \to \mathit{Stream}\ A$ (therefore only containing infinite streams), we can express it by the following definition. Note that coinductive records must be marked with a the coinductive flag, as shown.

```
record Stream (A : Set) : Set where
  coinductive
  field
    hd : A
    tl : Stream A
```

We can then define corecursive functions by copattern matching by defining how the value produced by the function reduces when eliminated through each one of the record's projections. For instance, in order to define $\mathit{natStream} : \mathbb{N} \to \mathit{Stream}\ \mathbb{N}$ we must explain how $\mathit{natStream}\ n$ reduces when we inspect the fields *hd* and *tl*. Note that, as the term $\mathit{natStream}\ (n + 1)$ does not reduced by itself, but only when eliminated through a projection, this system is terminating.

```
natStream : Nat → Stream-Nat
hd (natStream n) = n
tl (natStream n) = natStream (succ n)
```

In order to compare both presentations of coinduction, we can also look at how the *n*-th function can be defined in this setting. Note that as *n*-th is defined by recursion (in its first argument), and not by corecursion, it uses projections in a fundamentally different way from *natStream*. Whereas *natStream* uses them to explain how the value of the function reduces, *n*-th uses projections in order to access the fields of the stream *st*, which is given as second argument. Also note a very important change here, when comparing with the function *n*-th defined for musical coinduction: because with copattern matching coinduction we only have coinductive types with one constructor, which in this case is the constructor $\_ :: \_$, then all streams are infinite in this case, and thus this function can be total.

```
nth : Nat → Stream Nat → Nat
nth zero st = hd st
nth (succ n) st = nth n (tl st)
```

We remark that whereas both musical and copattern matching coinduction solve the problem of non-termination, when using copattern matching coinduction we do not have to deal with control operators, and thus we have a much more natural and user-friendly way of using coinduction.

A last remark is that, even though in this setting coinductive types are expressed as records, and we saw in subsection x that most records enjoy eta-conversion, this is disabled for coinductive records, as it can lead to non-termination issues. Indeed, if we were to try to eta-expand $\mathit{natStream}\ 0$ we would have the infinite unraveling

$$\mathit{natStream}\ 0 \longrightarrow \mathit{Record}\{\mathit{hd} = 0; \mathit{tl} = \mathit{natStream}\ 1\} \longrightarrow \mathit{Record}\{\mathit{hd} = 0; \mathit{tl} = \mathit{Record}\{\mathit{hd} = 1; \mathit{tl} = \mathit{natStream}\ 2\}\} \longrightarrow \dots,$$

where *Record* is the generic constructor for records.

Synctactic characterisation of productivity

When discussing the syntactic characterisation of coinduction in subsection x, we saw that the definition of coinductive functions had to follow a certain pattern. Namely, the body of the function must be directly equal to an instance on a constructor, which then should guard any corecursive call, as is the case in

$$\mathit{natStream}\ n = n :: (\mathit{natStream}\ (n + 1)).$$

This syntactic criterion is enforced by Agda in order to guarantee that the function is productive. For instance, the previous definition of *natStream* given on the setting of musical coinduction follows this criterion, but the following definition of the function *filter* on streams gets rejected, as the body of the first clause is not directly equal to an instance of a constructor.

```
filter : {A : Set} → (A → Bool) → Stream A → Stream A
filter f (hd :: tl) = if (f hd) then (hd :: ♯ (filter f (♭ tl))) else (filter f (♭ tl))
filter f [] = []
```

It is easy to see why this function cannot be accepted, as by supplying a function that is false everywhere and a stream which is infinite we get a calculation that never stops. However, this criterion is also sometimes too restrictive, and does not detect all productive functions. For instance, the following implementation of *natStream* is extensionally equal to the one previously presented, however it gets rejected by Agda because the corecursive call is not a direct argument to the constructor (we also say that it is not guarded by a constructor). In order to remedy this Agda also allows the use of *sized types* to show productivity, however as this is not the focus of this work we will not discuss them here.

```
natStream-fail : Nat → Stream Nat
natStream-fail n = n :: (if true then ♯ (natStream-fail (n+1)) else ♯ (natStream-fail (n+1)))
```

This syntactic criterion is also enforced when using copattern matching coinduction. However, as in this setting corecursive functions are defined through projections, the body of the function is already always equal to an instance of the only constructor. For instance, a corecursive definition producing a stream and defined by copattern matching as $hd\ (f\ \vec{x}) = y_{hd}$, $tl\ (f\ \vec{x}) = y_{tl}$ can always be reassembled into an instance of the constructor as $f\ \vec{x} = y_{hd} :: y_{tl}$. This rules out functions like *filter*, as we can see that there is no way of writing them in this setting.

Therefore, the only restriction in this case is that the term y_{tl} must start with a corecursive call. For instance, the definition of *natStream* that we gave when presenting copattern matching coinduction satisfies this criteria, as the clause defining the tail is defined by

$$tl\ (natStream\ n) = natStream\ (n + 1)$$

and thus starts with a corecursive call. However, by reusing the same idea as the one used on the implementation of *natStream-fail*, we can build the following function, which does not satisfy the criteria and is therefore rejected, even thought it is productive.

```
natStream-fail : Nat → Stream Nat
hd (natStream-fail n) = n
tl (natStream-fail n) = if true then (natStream-fail (n+1)) else (natStream-fail (n+1))
```

2.2 Coinduction in the λΠ-calculus modulo rewriting

In this subsection, we will show how the coinduction of Agda can be expressed in the λΠ-calculus modulo rewriting. We reuse the same conventions established in X to represent Dedukti definitions. In particular, we also ourselves once again to write some arguments as subscripts when this simplifies the notation, for instance we will write $\flat_A\ M$ as a simplification of $\flat\ A\ M$. Finally, we will use a superscript as in $A^{\text{♪}}$ when translating the musical version of A and as in A^{co} when translating the copattern matching version of A.

Musical coinduction

To encode musical coinduction, we first start by declaring constants for the control operators. Note that there is no constant for ♯ in the translation, as we will explain after.

$$\infty : U\ (Set\ z) \to U\ (Set\ z) \qquad (\infty\text{-decl})$$
$$\flat : \Pi A : U\ (Set\ z).El\ (\infty\ A) \to El\ A \ \text{(written as } \flat_A) \qquad (\flat\text{-decl})$$

In order to encode a coinductive type declaration, we proceed the same as when encoding an inductive type. We first declare a constant to define the type itself and then we declare constants to define the constructors of the type. For instance, to define the type *Stream* we define following constants, the first representing the declaration of the type, and the last two the declaration of the constructors.

$$\begin{array}{lr}
\mathit{Stream}^{\blacktriangleright} : U\ (\mathit{Set}\ z) \to U\ (\mathit{Set}\ z) & (\mathit{Stream}^{\blacktriangleright}\text{-decl}) \\
:: : \Pi A : U\ (\mathit{Set}\ z).\mathit{El}\ A \to \mathit{El}\ (\infty\ (\mathit{Stream}^{\blacktriangleright}\ A)) \to \mathit{El}\ (\mathit{Stream}^{\blacktriangleright}\ A)\ \text{(written infix as } ::_A) & (::\text{-decl}) \\
[] : \Pi A : U\ (\mathit{Set}\ z).\mathit{El}\ (\mathit{Stream}^{\blacktriangleright}\ A)\ \text{(written as } []_A) & ([]\text{-decl})
\end{array}$$

This generalizes the encoding of inductive types and is fairly straightforward, but the interesting part comes when encoding corecursive functions. If we were to define a constant $\sharp_A$, as done with $\flat_A$, and translate the definitions directly, we would have the following definition for $\mathit{natStream}^{\blacktriangleright}$.

$$\begin{array}{lr}
\mathit{natStream}^{\blacktriangleright} : \mathit{El}\ \mathbb{N} \to \mathit{El}\ (\mathit{Stream}^{\blacktriangleright}\ \mathbb{N}) & (\mathit{natStream}^{\blacktriangleright}\text{-decl}) \\
\mathit{natStream}^{\blacktriangleright}\ n \longrightarrow n ::_{\mathbb{N}} (\sharp_{\mathbb{N}}\ (\mathit{natStream}^{\blacktriangleright}\ (\mathit{succ}\ n))) & (\mathit{natStream}^{\blacktriangleright}\text{-red})
\end{array}$$

However, in this setting we cannot forbid reductions which happen under a $\sharp_A$ sign, as done in the semantics of Agda. This means that this rule causes non-termination, as we have the infinite chain

$$\mathit{natStream}^{\blacktriangleright}\ n \longrightarrow n ::_{\mathbb{N}} (\sharp_{\mathbb{N}}\ (\mathit{natStream}^{\blacktriangleright}\ (\mathit{succ}\ n))) \longrightarrow n ::_{\mathbb{N}} (\sharp_{\mathbb{N}}\ ((\mathit{succ}\ n) ::_{\mathbb{N}} (\sharp_{\mathbb{N}}\ (\mathit{natStream}^{\blacktriangleright}\ (\mathit{succ}\ (\mathit{succ}\ n)))))) \longrightarrow \ldots .$$

Therefore, we do not proceed like this. Instead, we apply the idea also used in the internal syntax representation of Agda to represent a function using two versions. In the case of *natStream* we have the following declarations.

$$\begin{array}{lr}
\mathit{natStream}^{\blacktriangleright} : \mathit{El}\ \mathbb{N} \to \mathit{El}\ (\mathit{Stream}^{\blacktriangleright}\ \mathbb{N}) & (\mathit{natStream}^{\blacktriangleright}\text{-decl}) \\
\sharp\text{-}\mathit{natStream}^{\blacktriangleright} : \mathit{El}\ \mathbb{N} \to \mathit{El}\ (\infty\ (\mathit{Stream}^{\blacktriangleright}\ \mathbb{N})) & (\sharp\text{-}\mathit{natStream}^{\blacktriangleright}\text{-decl})
\end{array}$$

The idea here is that each corecursive function will have its halted version, which we can use in corecursive calls without non-termination problems. In this setting, an application of $\sharp$ to a corecursive call is translated into the halted version of it. For instance, to finish the definition of $\mathit{natStream}^{\blacktriangleright}$ we declare the following rewrite rules.

$$\begin{array}{lr}
\mathit{natStream}^{\blacktriangleright}\ n \longrightarrow n::(\sharp\text{-}\mathit{natStream}^{\blacktriangleright}\ (\mathit{succ}\ n)) & (\mathit{natStream}^{\blacktriangleright}\text{-red}) \\
\flat_{\mathbb{N}}\ (\sharp\text{-}\mathit{natStream}^{\blacktriangleright}\ n) \longrightarrow \mathit{natStream}^{\blacktriangleright}\ n & (\sharp\text{-}\mathit{natStream}^{\blacktriangleright}\text{-red})
\end{array}$$

The first rewrite rule corresponds to the definition of the function, whereas the second allows to transform the halted version of the function into a computing one, through the $\flat$ symbol. Note that, as $\sharp\text{-}\mathit{natStream}^{\blacktriangleright}\ (\mathit{succ}\ n)$ does not reduce by itself, but only when under a $\flat$, these rewrite rules are terminating.

Copattern matching coinduction

Whereas the encoding of musical coinduction needs the duplication of function symbols in order to eliminate the $\sharp$ operator, we will see that copattern matching coinduction admits a much simple encoding, as we can obtain terminating corecursive definitions by just orienting the clauses defining corecursive functions.

Firs, in order to encode a coinductive type declaration, we declare once again a constant to represent the type. However, instead of declaring constants to define the constructors, we now declare constants to define the projections. The type Stream^{co} can for instance be represented by the following constants.

$$\begin{array}{lr}
\mathit{Stream}^{co} : U\ (\mathit{set}\ z) \to U\ (\mathit{set}\ z) & (\mathit{Stream}^{co}\text{-decl}) \\
\mathit{hd} : \Pi A : U\ (\mathit{set}\ z).\mathit{El}\ (\mathit{Stream}^{co}\ A) \to \mathit{El}\ A\ \text{(written as } \mathit{hd}_A) & (\mathit{hd}\text{-decl}) \\
\mathit{tl} : \Pi A : U\ (\mathit{set}\ z).\mathit{El}\ (\mathit{Stream}^{co}\ A) \to \mathit{El}\ (\mathit{Stream}^{co}\ A)\ \text{(written as } \mathit{tl}_A) & (\mathit{tl}\text{-decl})
\end{array}$$

Now corecursive functions can be translated by declaring a constant to represent the function and adding rewriting rules corresponding to the clauses. In the case of *natStream* we have the following declarations.

$$
\begin{array}{lr}
\mathit{natStream}^{co} : \mathit{El}\ \mathbb{N} \to \mathit{El}\ (\mathit{Stream}^{co}\ \mathbb{N}) & (\mathit{natStream}^{co}\text{-decl}) \\
\mathit{hd}_{\mathbb{N}}\ (\mathit{natStream}^{co}\ n) \longrightarrow n & (\mathit{natStream}^{co}\text{-red1}) \\
\mathit{tl}_{\mathbb{N}}\ (\mathit{natStream}^{co}\ n) \longrightarrow \mathit{natStream}^{co}\ (\mathit{succ}\ n) & (\mathit{natStream}^{co}\text{-red2})
\end{array}
$$

Note that, as no rewrite rule allows to reduce $\mathit{natStream}^{co}\ (\mathit{succ}\ n)$ by itself, the defined rewrite rules are terminating.

2.3 Examples on translating coinduciton

In order to understand how the encoding generalizes for other cases, we consider a series of examples.

n-th

We start with the function *n*-th, which was already discussed for both presentations of coinduction.

To translate the version using musical coinduction we first need to translate the inductive type constructor *Maybe*. We proceed as usual, declaring one constant to encode the type and one constant for each constructor.

$$
\begin{array}{lr}
\mathit{Maybe} : U\ (\mathit{Set}\ z) \to U\ (\mathit{Set}\ z) & (\mathit{Maybe}\text{-decl}) \\
\mathit{just} : \Pi A : U\ (\mathit{Set}\ z).\mathit{El}\ A \to \mathit{El}\ (\mathit{Maybe}\ A)\ \text{(written as } \mathit{just}_A) & (\mathit{just}\text{-decl}) \\
\mathit{nothing} : \Pi A : U\ (\mathit{Set}\ z).\mathit{El}\ (\mathit{Maybe}\ A)\ \text{(written as } \mathit{nothing}_A) & (\mathit{nothing}\text{-decl})
\end{array}
$$

Now, in order to encode *n*-th, we declare only one constant to represent the function, and then the expected rewrite rules. Even thought we saw that the translation of corecursive functions written with musical coinduction uses two constant declarations, *n*-th is not corecursive but recursive, and as such admits a straightforward representation with just one constant.

$$
\begin{array}{lr}
\mathit{n\text{-}th}^{\music} : \mathit{El}\ \mathbb{N} \to \mathit{El}\ (\mathit{Stream}^{\music}\ \mathbb{N}) \to \mathit{El}\ (\mathit{Maybe}\ \mathbb{N}) & (\mathit{n\text{-}th}^{\music}\text{-decl}) \\
\mathit{n\text{-}th}^{\music}\ \mathit{zero}\ (x ::_{\mathbb{N}} l) \longrightarrow \mathit{just}_{\mathbb{N}}\ x & (\mathit{n\text{-}th}^{\music}\text{-red1}) \\
\mathit{n\text{-}th}^{\music}\ (\mathit{succ}\ n)\ (x ::_{\mathbb{N}} l) \longrightarrow \mathit{n\text{-}th}^{\music}\ n\ (\flat_{\mathbb{N}}\ l) & (\mathit{n\text{-}th}^{\music}\text{-red2}) \\
\mathit{n\text{-}th}^{\music}\ \_\ []_{\mathbb{N}} \longrightarrow \mathit{nothing}_{\mathbb{N}} & (\mathit{n\text{-}th}^{\music}\text{-red3})
\end{array}
$$

In order to test the definitions of $\mathit{n\text{-}th}^{\music}$ and $\mathit{natStream}^{\music}$ we can try to compute the normal form of $\mathit{n\text{-}th}^{\music}\ 1\ (\mathit{natStream}^{\music}\ 6)$, where we write numerals in decimal notation to simplify the presentation. This gives the following rewrite sequence, which computes to the value $\mathit{just}_{\mathbb{N}}\ 7$ as expected.

$$
\begin{array}{rl}
\mathit{n\text{-}th}^{\music}\ 1\ (\mathit{natStream}^{\music}\ 6) \longrightarrow \mathit{n\text{-}th}^{\music}\ 1\ (6 ::_{\mathbb{N}} (\sharp\text{-}\mathit{natStream}^{\music}\ 7)) & (\mathit{natStream}^{\music}\text{-red}) \\
\longrightarrow \mathit{n\text{-}th}^{\music}\ 0\ (\flat_{\mathbb{N}}\ (\sharp\text{-}\mathit{natStream}^{\music}\ 7)) & (\mathit{n\text{-}th}^{\music}\text{-red2}) \\
\longrightarrow \mathit{n\text{-}th}^{\music}\ 0\ (\mathit{natStream}^{\music}\ 7) & (\flat\text{-red}) \\
\longrightarrow \mathit{n\text{-}th}^{\music}\ 0\ (7 ::_{\mathbb{N}} (\sharp\text{-}\mathit{natStream}^{\music}\ 8)) & (\mathit{natStream}^{\music}\text{-red}) \\
\longrightarrow \mathit{just}_{\mathbb{N}}\ 7 & (\mathit{n\text{-}th}^{\music}\text{-red1})
\end{array}
$$

We can also look at the translation of *n-th* when using copattern-matching coinduction.

$$
\begin{array}{lr}
\mathit{n\text{-}th}^{co} : \mathit{El}\ \mathbb{N} \to \mathit{El}\ (\mathit{Stream}^{co}\ \mathbb{N}) \to \mathit{El}\ \mathbb{N} & (\mathit{n\text{-}th}^{co}\text{-decl}) \\
\mathit{n\text{-}th}^{co}\ \mathit{zero}\ x \longrightarrow \mathit{hd}_{\mathbb{N}}\ x & (\mathit{n\text{-}th}^{co}\text{-red1}) \\
\mathit{n\text{-}th}^{co}\ (\mathit{succ}\ n)\ x \longrightarrow \mathit{n\text{-}th}^{co}\ n\ (\mathit{tl}_{\mathbb{N}}\ x) & (\mathit{n\text{-}th}^{co}\text{-red2})
\end{array}
$$

Once again, we can try to compute the normal form of $\mathit{n\text{-}th}^{co}\ 1\ (\mathit{natStream}^{co}\ 6)$ to see how the computation behaves.

$$\begin{aligned}
\mathit{n\text{-}th}^{co}\ 1\ (\mathit{natStream}^{co}\ 6) &\longrightarrow \mathit{n\text{-}th}^{co}\ 0\ (\mathit{tl}_{\mathbb{N}}\ (\mathit{natStream}^{co}\ 6)) && (\mathit{n\text{-}th}^{co}\text{-red2})\\
&\longrightarrow \mathit{n\text{-}th}^{co}\ 0\ (\mathit{natStream}^{co}\ 7) && (\mathit{natStream}^{co}\text{-red2})\\
&\longrightarrow \mathit{hd}_{\mathbb{N}}\ (\mathit{natStream}^{co}\ 7) && (\mathit{n\text{-}th}^{co}\text{-red1})\\
&\longrightarrow 7 && (\mathit{natStream}^{co}\text{-red1})
\end{aligned}$$

$$\mathit{n\text{-}th}\ 1\ (\mathit{natStream}\ 6) \longrightarrow \mathit{n\text{-}th}\ 1\ (6 ::_{\mathbb{N}} (\sharp\text{-}\mathit{natStream}\ 7)) \longrightarrow \mathit{n\text{-}th}\ 0\ (\mathit{natStream}\ 7) \longrightarrow \mathit{n\text{-}th}\ 0\ (7 ::_{\mathbb{N}} (\mathit{natStream}\ 8))\,.$$

3 Practical Implementation

In the previous section, we saw how coinduction can be used in Agda and how such definitions can be translated in the $\lambda\Pi$-calculus modulo rewriting. We now detail the practical details of how this translation is implemented in Agda2Dedukti.

A large part of this internship was also dedicated to restarting the development of Agda2Dedukti, which was halted since September of 2020, when Guillaume Genestier had to stop its development to write his PhD thesis. Therefore, we also detail many other contributions that were made for improving it.

3.1 Translation of Coinduction

As already mentioned, the proposed encoding of coinduction in the $\lambda\Pi$-calculus modulo rewriting is based on the representation of the internal syntax of Agda. Therefore, the main challenge here was adapting Agda2Dedukti to correctly translate the internal representations into Dedukti, something that was not done in Genestier's prototype. The main problem the original prototype had and which prevented the translation of coinduction lied on the translation of clauses defining corecursive functions.

A clause defining a normal recursive function f is generally of the form $f\ \vec{x} = y$, and therefore Agda internally represents a clause by a head symbol f, by a list of applied patterns $\vec{x}$ and a body y. However, when we were discussing the encoding of coinduction we saw that corecursive clause definitions do not always satisfy this criterion.

In the case of musical coinduction, each function declaration f is duplicated into a halted version $\sharp\text{-}f$. The clauses of f itself are of the form $f\ \vec{x} = y$, however the only clause defining $\sharp\text{-}f$ is

$$\flat\ (\sharp\text{-}f\ \vec{x}) = f\ \vec{x}\,.$$

In order to represent such a clause in the form $f\ \vec{x} = y$, Agda puts projections in *postfix form*. This means the previous clause is represented internally as $\sharp\text{-}f\ \vec{x}\ .\flat = f\ \vec{x}$, where the dot in $.\flat$ means that this application should be translated in prefix form.

This happens even more frequently when translating copattern matching corecursive functions. Indeed, a corecursive function f defined by copattern matching will have its clauses in the form $\pi\ (f\ \vec{x})\ \vec{z} = y$, which is then represented in the form $f\ \vec{x}\ .\pi\ \vec{z} = y$. For instance, the first clause defining the function *natStream* was represented internally as $\mathit{natStream}\ n\ .\mathit{hd} = n$.

As Genestier's prototype did not take this into account, such functions declarations were translated in an incorrect way. For instance, the first clause of *natStream* was being translated as

$$\mathit{natStream}\ n\ \mathit{hd} \longrightarrow n\,,$$

whose left side is not even well typed, as $\mathit{natStream}\ n$ is of type $\mathit{Stream}\ \mathit{Nat}$ and hd is of type $\{A : \mathit{Set}\} \to \mathit{Stream}\ A \to A$.

Therefore, in order to correctly translate these clauses we had to change the code implementing the translation of clauses in order to properly account for this representation. As the original code was not documented, this turned out to be not so trivial. Moreover, when representing a clause $f\ \vec{x} = y$ Agda does not store internally all the type information of

the left-hand side, but only of the head symbol f, this information needs to be reconstructed while translating patterns, which makes the process a bit tricky.

Finally, we also had to adapt other parts of the translator which did not interact properly with coinduction. Explicitly, the translation of eta-expansion (see details on Ap b) was previously treating all records in an homogeneous way, meaning that even coinductive records were translated with eta-expansion in Dedukti. However, we saw in section x that eta-expansion causes non-termination when added to coinductive records. This did not cause a problem before, as coinduction was not a feature supported by the translator, but as this is now the case we had to adapt the translation in order to only translate with eta-expansion the records which also have it in Agda.

This implementation allowed us to translate automatically multiple Agda files containing coinductive definitions and corecursive functions into Dedukti. Some of these translated files can be found in INSERT-NEW-GIT-REPO-WITH-TRANSLATIONS.

3.2 Agda2Lambdapi

We had previously discussed that, in order to use the $\lambda\Pi$-calculus modulo rewriting in practice, researchers at Deducteam have developed two implementations Dedukti and Lambdapi. Whereas Dedukti will probably be discontinued, Lambdapi is under active development and extends it in multiple ways. Lambdapi features most notably interactive proof development, with a proof mode and tactics, but also other features such as efficient rewriting, metavariables and implicit arguments. As the prototype translator Agda2Dedukti was only capable of translating into Dedukti, and not Lambdapi, the extension of the translator with an Agda2Lambdapi mode was a natural goal of the internship, which would allow the translation of files into both Dedukti and Lambdapi.

Most of the development of this new mode consisted of adapting the syntax of the output. Moreover, Lambdapi files must state explicitly in their beginning which other files they use, something which was not needed in Dedukti, and therefore the translation of each file now needed to take this into account. However, the most challenging part of this development was dealing with AC symbols, which are used to implement the translation of universe polymorphism.

Dealing with AC symbols

Associative commutative (or just AC) symbols, are symbols which satisfy an associative commutative equational theory. For instance, if we declare $+ : \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}$ as being AC, then we automatically get the conversions $x + (y + z) \equiv (x + y) + z$ and $x + y \equiv y + x$. Note that, even though we could add the conversion $x + (y + z) \equiv (x + y) + z$ by means of the rewriting rule

$$x + (y + z) \longrightarrow (x + y) + z\,,$$

there is no way to add the conversion relation $x + y \equiv y + x$ with a terminating rewrite system. Therefore, the ability of having AC symbols strictly enriches the capability of handling equational theories. AC symbols may also enjoy a richer type of matching, called AC matching. In the presence of AC matching, all terms of the form $x + x$, $x + (y_1 + x)$, $x + (y_1 + (y_2 + x))$, etc are matched by the rule

$$x + x \longrightarrow x\,,$$

whereas with normal matching we would have to declare a rewrite rule

$$x + (y_1 ... (y_k + x) ...) \longrightarrow x + (y_1 ... (y_{k-1} + y_k) ...)\,.$$

for each k (thus, an infinity of rewrite rules).

We will not enter in all the details here, but the point is that the implementation of universe levels used both AC symbols and AC matching, which is particularly needed when using universe polymorphism. However, whereas Dedukti featured both AC symbols and AC matching, Lambdapi does not feature AC matching, as its implementation is complex and error-prone. Therefore, the challenge here was adapting the representation of universe levels such that AC matching was not required. In order to do so, Frédéric Blanqui introduced in Lambdapi a mechanism to put terms in a canonical form internally, such that given an order on variable names, the canonical form uses the AC identities to order them.

The precise order of the variables is irrelevant, the interesting point is that the term $x + (y_1 ... (y_k + x) ...)$ will be represented either as $x + (x + (...))$ (case I) or $y_{i_1} + (...(x + (x + ...))...)$ (case II) or $y_{i_1} + (...(y_{i_k} + (x + x))...)$ (case III), and thus the

occurrences of x will be grouped together. This allows us to replace the rule $x + x \longrightarrow x$ using AC matching by the two rules

$$x + x \longrightarrow x$$

matching (III) and

$$x + (x + y) \longrightarrow x + y$$

matching (I) and (II).

Using this technique, we modified with Frédéric Blanqui the encoding of universe levels in order to get rid of AC matching. This also evolved the testing of the canonical form mechanism and the reporting of multiple bugs to the development of Lambdapi. This encoding allowed us then to correctly translate multiple Agda files using universe polymorphism into Lambdapi.

3.3 Adding support for latest Agda version

4 Future Work

4.1 Universe polymorphism beyond prenex

4.2 Interoperability between predicative and impredicative type theory

5 Conclusion

Appendix A References

[1] A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: Programming infinite structures by observations. 2013.

[2] A. Assaf. *A framework for defining computational higher-order logics*. Theses, École polytechnique, Sept. 2015.

[3] A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. Dedukti: a logical framework based on the $\lambda\,\pi$-calculus modulo theory. Manuscript, 2016.

[4] F. Blanqui, G. Dowek, É. Grienenberger, G. Hondet, and F. Thiré. Some axioms for mathematics. In N. Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina* (*Virtual Conference*), volume 195 of *LIPIcs*, pages 20:1–20:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[5] D. Cousineau and G. Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In S. R. Della Rocca, editor, *Typed Lambda Calculi and Applications*, pages 102–117, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[6] G. Dowek and F. Thiré. Logipedia: a multi-system encyclopedia of formal proofs. Manuscript.

[7] G. Genestier. Encoding agda programs using rewriting. 2020.

[8] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, Jan. 1993.

[9] G. Hondet and F. Blanqui. Encoding of Predicate Subtyping with Proof Irrelevance in the $\lambda\Pi$-Calculus Modulo Theory. In U. de'Liguoro, S. Berardi, and T. Altenkirch, editors, *26th International Conference on Types for Proofs and Programs* (*TYPES 2020*), volume 188 of *Leibniz International Proceedings in Informatics* (*LIPIcs*), pages 6:1–6:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[10] nLab authors. eta-conversion. `http://ncatlab.org/nlab/show/eta-conversion`, July 2021. Revision 12.

[11] F. Thiré. Sharing a library between proof assistants: Reaching out to the HOL family. In F. Blanqui and G. Reis, editors, *Proceedings of the 13th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP@FSCD 2018, Oxford, UK, 7th July 2018*, volume 274 of *EPTCS*, pages 57–71, 2018.

[12] F. Thiré. *Interoperability between proof systems using the logical framework Dedukti*. PhD thesis, ENS Paris-Saclay, 2020.

Appendix B Typing rules for the $\lambda\Pi$-calculus modulo rewriting

The following describes typing in the $\lambda\Pi$-calculus modulo rewriting for a given set of rewrite rules $\mathcal{R}$. Given a context Γ, a signature Σ and $M, A \in \Lambda_{\lambda\Pi}$, we define the typing judgment $\Sigma; \Gamma \vdash M : A$ inductively by the following deduction rules[4]. The relation $\equiv$ in the rule Conv is the least equivalence relation containing $\equiv_\beta$ and the context and substitution

closure of the rules in $\mathcal{R}$. In the rules Decl, Prod, Convs, Abs and Conv, the letter s stands either for **Type** or **Kind**.

Context forming rules

$$\frac{}{\Sigma;\emptyset \text{ well-formed}}\ \text{Empty} \qquad \frac{\Sigma;\Gamma \vdash A : s \qquad x \notin \Gamma}{\Sigma;\Gamma, x : A \text{ well-formed}}\ \text{Decl}$$

Term forming rules

$$\frac{\Sigma;\Gamma \text{ well-formed}}{\Sigma;\Gamma \vdash \textbf{Type} : \textbf{Kind}}\ \text{Sort}$$

$$\frac{\Sigma;\Gamma \vdash A : \textbf{Type} \qquad \Sigma;\Gamma, x : A \vdash B : s}{\Sigma;\Gamma \vdash \Pi x : A.B : s}\ \text{Prod}$$

$$\frac{\Sigma;\Gamma \text{ well-formed} \qquad x : A \in \Gamma}{\Sigma;\Gamma \vdash x : A}\ \text{Var}$$

$$\frac{\Sigma;\Gamma \text{ well-formed} \qquad c : A \in \Sigma \qquad \Sigma;\emptyset \vdash A : s}{\Sigma;\Gamma \vdash c : A}\ \text{Cons}$$

$$\frac{\Sigma;\Gamma \vdash \Pi x : A.B : s \qquad \Sigma;\Gamma, x : A \vdash M : B}{\Sigma;\Gamma \vdash \lambda x : A.M : \Pi x : A.B}\ \text{Abs}$$

$$\frac{\Sigma;\Gamma \vdash M : \Pi x : A.B \qquad \Sigma;\Gamma \vdash N : A}{\Sigma;\Gamma \vdash MN : B(N/x)}\ \text{App}$$

Conversion rule

$$\frac{\Sigma;\Gamma \vdash M : A \qquad \Sigma;\Gamma \vdash B : s \qquad A \equiv B}{\Sigma;\Gamma \vdash M : B}\ \text{Conv}$$

Typing rules for the λΠ-calculus modulo rewriting

Appendix C End-notes

You can click on the number to get back to where the end-note was made.

[1] Its only use is to give a type to **Type** and to terms of the form $A_1 \to A_2 \to \ldots \to \textbf{Type}$

[2] More precisely, the addition of dependent types only renders the system more expressive because we are also allowed to have type families, otherwise it would be always possible to replace any $\Pi x : A.B$ by $A \to B$.

[3] More precisely, with minimal intuitionistic predicate logic, that is, the fragment on intuitionistic predicate logic only featuring implication and universal quantification.

[4] Such an encoding is also called deep, as implication and universal quantification is represented by new added constants, and not by the Π and the abstraction of the $\lambda\Pi$-calculus.

[5] The term *sort* here means something else than the sorts of a type system.

[6] This part only concerns *canonical terms*, that is, closed terms in normal form. For instance, if we consider elements of $\mathbb{N}$ closed but not in normal form we also have $(\lambda x : \mathbb{N}.x)\ 0$ which does not fit this description. Likewise, if we consider elements of $\mathbb{N}$ in normal form but not closed, we can have for instance a variable x of type $\mathbb{N}$. However, by imposing those two constraints at the same time, we are assured (in systems which have the *canonicity* property, which is a desirable metaproperty in most cases) that such elements of inductive types are indeed the least fixed points of the presented function.

[7] However, this feature can be enabled by a flag.

[8] A hierarchy *Prop* of proof irrelevant types was recently added to Agda, however the "standard" way of doing Agda is to do everything with *Set*. For instance, Agda's standard library do not use *Prop*.

[9] We can also define η-equivalence by means of η-reduction, however this is not so well-behaved when dealing with other types, such as the singleton type[10].

[10] See the Agda documentation for a detailed description of which records are allowed or note to feature η-equivalence