Representing Agda and Coinduction on the λΠ-calculus modulo rewriting

Thiago Felicissimo

[1] t

[2] t

1 Introduction

The problem of proof interoperability

TODO: more direct introduction, talk about why translate agda and coinduction is important. remove section introduction and put everything on the fiche de synthese.

In light of the *Curry-Howard* correspondence – also known as the propositions as types correspondence – we experience an emergence of a technology which is leading us into a new era of mathematics. Proof assistants, which put simply are programming languages for proving, are increasingly being used on the development and verification of formal proofs. 20 years after its original failure, we are finally getting closer to the objectives of the QED manifesto, of building a mathematical library of automatically checked proofs.

For instance, 97% of the famous The Hundred Greatest Theorems list has already been formalized in some proof assistant. However, many of these theorems are only available in a some systems, and a user wanting to use the result on a different one has no option other then redoing the proof, a very time and resource demanding task.

Quite naturally, the already well known problem of interoperability of programs is transported by the Curry-Howard correspondence to a problem of interoperability of proofs. However, this question is even amplified: proof assistants, being used as logics, need a much intricate set of features, such as complex type systems, inductive types,

Therefore, this question is arguably one of the central ones when

2 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 which of the features of the proof assistant it is able to translate.

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

Starting point: a λ-calculus with dependent types

The lambda-calculus with dependent types, or $\lambda\Pi$-calculus, was proposed by [7] as a logical framework in which many proof and type systems can be expressed. Its syntax is given by

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

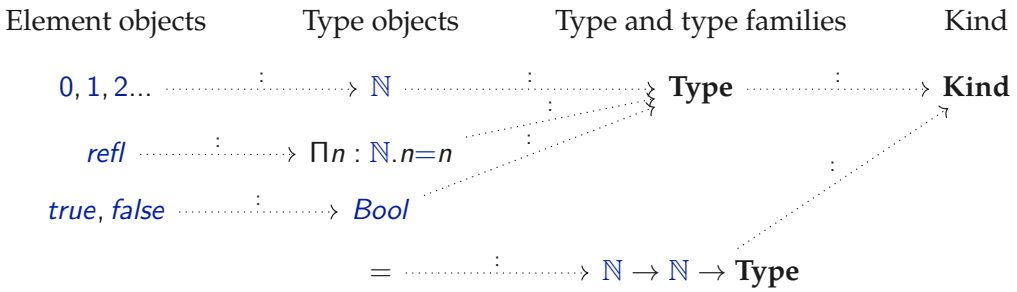
where $\mathcal{C}$ is an infinite set of constants and $\mathcal{X}$ is an infinite set of variables. We denote $\Lambda_{\lambda\Pi}$ the set of terms generated by this grammar. Conversion is defined as usual by β-equivalence, and we write $\Pi x : A.B$ as $A \to B$ when x does not appear in B.

A *context* Γ is a finite set 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 set of pairs $c : A$, where c is a constant, $A \in \Lambda_{\lambda\Pi}$ and every constant can only appear once. We write constants which compose the signature Σ in blue, as they will be much used when declaring encodings or theories. Typing on the $\lambda\Pi$-calculus is defined through judgements 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, the type system can be separated into type objects (terms typed by **Type**) and element 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 element 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 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 on the proposition as types approach we have that $\lambda x : P.x$ is a proof of $P \Rightarrow P$, on the judgment as types this is expressed by the term $\Rightarrow_{in} P\ P\ (\lambda x : P.x)$.

The judgment as types approach was the one originally proposed by [7] 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 feature that are orthogonal to those of the $\lambda\Pi$-calculus, such as System F. This can seem very strange, 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 see an important point we did not 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 $\Rightarrow_{el} P\ P\ \alpha_P\ (\Rightarrow_{in} P\ P\ (\lambda x : P.x))$. 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 $\Rightarrow_{el} P\ P\ \alpha_P\ (\Rightarrow_{in} P\ P\ (\lambda x : P.x))$ is stuck and cuts do not reduce anymore.

As pointed out by Assaf[1], encodings such as this one, which lack the preservation of computation, fail to be sound for more higher-order systems, such as the Calculus of Constructions. 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 on the $\lambda\Pi$-calculus

In 2007, Dowek and Cousineau proposed an extension of the $\lambda\Pi$-calculus in which the notion of computation can be extended by adding rewriting rules[4]. The syntax and the typing rules are kept the same, however we 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 $\equiv$ on 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 $\mathcal{R}$.

By addressing the poorness of computation on 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 [4], Dowek and Cousineau showed that we can express any functional PTS in a sound and complete way, which was already not possible on the $\lambda\Pi$-calculus. Since then, researchers at Deducteam have built on top of this work and proposed encoding of much richer features on the $\lambda\Pi/\mathcal{R}$-calculus, like inductive types[2], universe polymorphism[6], cumulativity[1][11], proof-irrelevance[8], etc.

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

Let's now take a second try at doing a judgment as types encoding of predicate logic, but now using rewrite rules. On 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 on the $\lambda\Pi/\mathcal{R}$-calculus: we can just declare a rewrite rule

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

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. We already have declared

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

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 $\mathit{El} : \iota \to \mathbf{Type}$, associating to each sort of the language a type of its elements.

$$\begin{aligned}
&\mathit{Set} : \mathbf{Type} && (\mathit{Set}\text{-decl})\\
&\iota : \mathit{Set} && (\iota\text{-decl})\\
&\mathit{El} : \mathit{Set} \to \mathbf{Type} && (\mathit{El}\text{-decl})
\end{aligned}$$

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 of A and returning a proof of $P\ x$. We are finished and we can show that this set of constants and rewrite rules provides a sound and complete encoding of many-sorted (minimal intuitionistic) predicate logic.

Actually, in [3] researchers at Deducteam proposed a theory, containing this 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[11][10][1], and most notably the encyclopedia of formal proofs expressed in Dedukti *Logipedia*[5] is one of the main projects at Deducteam.

3 Coinduction

A dual to induction

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 types $\mathit{List}\ \mathbb{N}$ 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 $\mathit{Stream}\ \mathbb{N}$.

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) 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 a constructor. Therefore, whereas induction is about building things, coinduction is about destructing them. We will see on the next part that when talking about 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.

The induction and coinduction principles

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 $\alpha_c : 1 + \mathbb{N} \times c \to c$ and morphisms in $\mathbf{Set}_F(\alpha_c, \alpha_d)$ are functions $f : c \to d$ making the following diagram commute.

$$\begin{array}{ccc} 1 + \mathbb{N} \times c & \xrightarrow{Ff} & 1 + \mathbb{N} \times d \\ \downarrow{\scriptstyle \alpha_c} & & \downarrow{\scriptstyle \alpha_d} \\ c & \xrightarrow{f} & d \end{array}$$

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 $\alpha_c : 1 + \mathbb{N} \times c \to c$, there is a unique function $\mathit{rec}(\alpha_c)$ making the following diagram commute.

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

We see the induction principle naturally arise.

Induction Principle: To define a function $\mathit{rec}\ \alpha_c : \mathit{List}\ \mathbb{N} \to c$, it suffices to define a function $\alpha_c : 1 + \mathbb{N} \times c \to c$.

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

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

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

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

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 $\beta_c : c \to 1 + \mathbb{N} \times c$ and morphisms in $\mathbf{Set}^F(\beta_c, \beta_d)$ are functions $f : c \to d$ making the following diagram commute.

$$\begin{array}{ccc} c & \xrightarrow{\quad f \quad} & d \\ \downarrow{\scriptstyle \beta_c} & & \downarrow{\scriptstyle \beta_d} \\ 1 + \mathbb{N} \times c & \xrightarrow{Ff} & 1 + \mathbb{N} \times d \end{array}$$

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 $\beta_c : c \to 1 + \mathbb{N} \times c$, there is a unique function $\mathit{corec}(\alpha_c)$ making the following diagram commute, thus yielding the coinduction principle.

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

Coinduction Principle: To define a function $c \to \mathit{Stream}\ \mathbb{N}$, it suffices to define a function $c \to 1 + \mathbb{N} \times c$.

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 also assures us that $\beta_{\mathit{Stream}\ \mathbb{N}}$ is an isomorphism, allowing us to write $\mathit{corec}\ \beta_c = \beta^{-1}_{\mathit{Stream}\ \mathbb{N}} \circ F(\mathit{corec}\ \beta_c) \circ \beta_c$. However, in the current setting we cannot give a as nice presentation[7] as in the case of pattern matching. However, we can change things if we impose that the coinductive type must have a single constructor. Here we get rid of the constructor [] (effectively imposing all streams to be infinite), leading us to the following diagram.

$$\begin{array}{ccc} c & \xrightarrow{\quad \mathit{corec}\ \beta_c \quad} & \mathit{Stream}\ \mathbb{N} \\ \downarrow{\scriptstyle \beta_c} & & \downarrow{\scriptstyle \beta_{\mathit{Stream}\ \mathbb{N}}} \\ \mathbb{N} \times c & \xrightarrow{F(\mathit{corec}\ \beta_c)} & \mathbb{N} \times (\mathit{Stream}\ \mathbb{N}) \end{array}$$

New Coinduction Principle: To define a function $\mathit{corec}\ \beta_c : c \to \mathit{Stream}\ \mathbb{N}$, it suffices to define a function $\beta_c : c \to \mathbb{N} \times c$, that is, two functions $\beta^1_c : c \to \mathbb{N}$ and $\beta^2_c : c \to c$.

Now we can define the function β_c not by making a case distinction on the constructors for the type, but rather for the arguments of its only constructor (also called the fields). The equality $\mathit{corec}\ \beta_c = \beta^{-1}_{\mathit{Stream}\ \mathbb{N}} \circ F(\mathit{corec}\ \beta_c) \circ \beta_c$ now gives

the following equations.

$$\begin{aligned} &\mathit{head}\ (\mathit{corec}\ \beta_c)\ x = \beta_c^1\ x \\ &\mathit{tail}\ (\mathit{corec}\ \beta_c)\ x = (\mathit{corec}\ \beta_c)\ (\beta_c^2\ x) \end{aligned}$$

This form of presenting coinduction is called copattern matching and was introduce by Abel et al in x. This presentation, which exists only for one constructor coinductive types, restores a symmetry and gives a much nicer way of using coinductive types. Let's look at an example of how to use them. If we take $c = *$, $\beta_*^1 : * \mapsto 0$ and $\beta_*^2 : * \mapsto *$ we get the following function, which builds the stream of only zeros.

$$\begin{aligned} &\mathit{head}\ (\mathit{corec}\ \beta_c) = 0 \\ &\mathit{tail}\ (\mathit{corec}\ \beta_c) = \mathit{corec}\ \beta_c \end{aligned}$$

The presentations of primitive (co)induction by (co)pattern matching also allows us to see how recursion and corecursion are characterized syntacticly. When defining $\mathit{rec}\ \alpha_c$ by recursion, the definition $(\mathit{rec}\ \alpha_c)\ (n :: l)$ can only be a function of n and $(\mathit{rec}\ \alpha_c)\ l$, that is, a recursive call can only be made on an argument of the constructor. On the other hand, when defining $\mathit{corec}\ \beta_c$ by corecursion, on the definition of $\mathit{tail}\ (\mathit{corec}\ \beta_c)\ x$ the corecursive call to $\mathit{corec}\ \beta_c$ must be outermost, as shown by the equation $\mathit{tail}\ (\mathit{corec}\ \beta_c)\ x = (\mathit{corec}\ \beta_c)\ (\beta_c^2\ x)$.

3.1 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 Lof Type Theory with many features, such as (co) inductive types, universe polymorphism, sized types, etc. On the following, we present some of its particularities, which are not present in all proof assistants.

A logic made unique by its type system

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 $A : \mathit{Set}$) or a proof of $A \Rightarrow A$ (if $A : \mathit{Prop}$), 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.

Most proof assistants use the separation between types and propositions to treat them differently. More precisely, they normally implement a flavor of *higher-order logic*, allowing propositions to be quantified over any domains. Explicitly, if P is a propositions and A is any type, we can always build $\forall_A x.P$ and its type will always be Prop, even when $A = \mathit{Prop}$. This is expressed by the deduction rule

$$\frac{\Sigma; \Gamma, x : A \vdash B : \mathit{Prop}}{\Sigma; \Gamma \vdash \Pi x : A.B : \mathit{Prop}}$$

This means that we can build propositions that talk about all propositions — including themselves! —, such as $\forall_{\mathit{Prop}}.p \Rightarrow p$.

The ability to build such "weird" propositions is refereed to as *impredicativity*, and it is a whole subject in its own. However, the important point to know is that, whereas propositions are usually treated impredicatively, normal types are treated predicatively. More explicitly, for normal types we usually have a stratified infinite hierarchy $\mathit{Set}_0 : \mathit{Set}_1 : \mathit{Set}_2 \ldots$ and the deduction rule

$$\frac{\Sigma; \Gamma \vdash A : \mathit{Set}_i \qquad \Sigma; \Gamma, x : A \vdash B : \mathit{Set}_j}{\Sigma; \Gamma \vdash \Pi x : A.B : \mathit{Set}_{max\{i,j\}}}$$

This implies that the type of $\Pi a : \mathit{Set}_0.a \to a$ is not Set_0, but Set_1. Therefore, by quantifying over all elements of a certain universe Set_i we move to a higher one. As we can see, predicative systems feature a more restricted type of quantification, which avoids building objects that self-reference, such as $\forall_{\mathit{Prop}} p.p \Rightarrow p$.

As Agda does not separate propositions and treats them directly as types, this means that propositions are also treated predicatively. Indeed, the proposition (or the type) $P \to P$ for all propositions (or types) P is not directly representable in Agda , as it is not possible to talk about all the types that live in all the Set_i. However, if we restrict quantification to Set_0, this can be expressed by $\Pi P : Set_0.P \to P$, which now lives in Set_1.

All in all, Agda has a logic that is very different from the ones of most proof assistants, in which propositions are treated as normal types and live in a predicative world.

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 checker.

For instance, we can declare the inductive type of natural numbers by

```
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 Lof 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 a function $List : Set \to Set$ which associates to each type A in Set the type of $List\ A$ of lists of elements in A. However, Set in Agda is just an alias for Set_0, as we have an infinite hierarchy $Set_0 : Set_1 : ...$ of sorts. Thus if we have an type B which lives in Set_1 we need to declare another function $List_1 : Set_1 \to Set_1$ 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 term

$$List : \Pi i : Level.Set_i \to Set_i\ .$$

which can then be instantiated at each level, avoiding the declaration of an infinite number of versions of $List$.

Such kind of polymorphism is called *prenex* as the level quantification occurs in the outer part of the term. However, Agda also supports a less used form of universe polymorphism called *non-prenex*, in which level quantification can appear anywhere in the term. This allows for instance to build a function

$$comp := \lambda a : (\Pi i : Level.Set_i).\lambda f : (\Pi i : Level.Set_i \to Set_i).\lambda i : Level.(f\ i)\ (a\ i)\ .$$

of type $(\Pi i : Level.Set_i) \to (\Pi i : Level.Set_i \to Set_i) \to \Pi i : Level.Set_i$ which applies a universe polymorphic function f to a universe polymorphic value a to get another universe polymorphic value.

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]

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

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 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.

Coinductive types

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

5 Future Work

5.1 Universe polymorphism beyond prenex

5.2 Interoperability between predicative and impredicative type theory

5.3 A second look at Inductive Types

Appendix A References

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

[2] 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.

[3] 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.

[4] 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.

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

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

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

[8] 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.

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

[10] 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.

[11] 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 on 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[3]. The relation $\equiv$ on 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 type families (types 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] You can try for yourself: when going through the diagram, as the function β_c has a codomain which is a sum, we would need to make a distinction whether we take the left or the right codomain. It's not clear how to give a clean presentation of this, as in the case for pattern matching.

[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[9].

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