The Yoneda Reduction of Polymorphic Types

Paolo Pistone
DISI, University of Bologna, Italy
paolo.pistone2@unibo.it

Luca Tranchini
Wilhelm-Schickard-Institut, Universität Tübingen, Germany
luca.tranchini@gmail.com

Abstract

In this paper we explore a family of type isomorphisms in System F whose validity corresponds, semantically, to some form of the Yoneda isomorphism from category theory. These isomorphisms hold under theories of equivalence stronger than $\beta\eta$-equivalence, like those induced by parametricity and dinaturality. We show that the Yoneda type isomorphisms yield a rewriting over types, that we call Yoneda reduction, which can be used to eliminate quantifiers from a polymorphic type, replacing them with a combination of monomorphic type constructors. We establish some sufficient conditions under which quantifiers can be fully eliminated from a polymorphic type, and we show some application of these conditions to count the inhabitants of a type and to compute program equivalence in some fragments of System F.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Program semantics

Keywords and phrases System F, Type isomorphisms, Yoneda isomorphism, Program equivalence

Digital Object Identifier 10.4230/LIPIcs.CSL.2021.35

Related Version A full version of the paper is available at `https://arxiv.org/abs/1907.03481`.

Funding *Paolo Pistone*: ERC CoG 818616 "DIAPASoN".
Luca Tranchini: DFG TR1112/4-1 "Falsity and Refutation. On the negative side of logic".

1 Introduction

The study of type isomorphisms is a fundamental one both in the theory of programming languages and in logic, through the well-known *proofs-as-programs* correspondence: type isomorphisms supply programmers with transformations allowing them to obtain simpler and more optimized code, and offer new insights to understand and refine the syntax of type- and proof-systems.

Roughly speaking, two types A, B are isomorphic when one can transform any call by a program to an object of type A into a call to an object of type B, without altering the behavior of the program. Thus, type isomorphisms are tightly related to theories of *program equivalence*, which describe what counts as the observable behavior of a program, so that programs with the same behavior can be considered equivalent.

The connection between type isomorphisms and program equivalence is of special importance for polymorphic type systems like System F (hereafter $\Lambda 2$). In fact, while standard $\beta\eta$-equivalence for $\Lambda 2$ and the related isomorphisms are well-understood [10, 12], stronger notions of equivalence (as those based on *parametricity* or *free theorems* [37, 19, 1]) are often more useful in practice but are generally intractable or difficult to compute, and little is known about the type isomorphisms holding under such theories.

$$\forall X.X \Rightarrow X \Rightarrow A \equiv A[X \mapsto 1+1] \qquad (*)$$
$$\forall X.(A \Rightarrow X) \Rightarrow (B \Rightarrow X) \Rightarrow C \equiv C[X \mapsto \mu X.A+B] \qquad (**)$$
$$\forall X.(X \Rightarrow A) \Rightarrow (X \Rightarrow B) \Rightarrow D \equiv D[X \mapsto \nu X.A \times B] \qquad (***)$$

Figure 1 Other examples of Yoneda type isomorphisms, where X only occurs positively in A, B, C and only occurs negatively in D.

$$\begin{aligned}
&\forall X.X \Rightarrow X \Rightarrow \forall Y.(\forall Z.(Z \Rightarrow X) \Rightarrow (\forall W.(W \Rightarrow Z) \Rightarrow W \Rightarrow X) \Rightarrow Z \Rightarrow Y) \Rightarrow (X \Rightarrow Y) \Rightarrow Y \\
&\overset{(*)}{\equiv} \forall Y.(\forall Z.(Z \Rightarrow 1+1) \Rightarrow (\forall W.(W \Rightarrow Z) \Rightarrow W \Rightarrow 1+1) \Rightarrow Z \Rightarrow Y) \Rightarrow (1+1 \Rightarrow Y) \Rightarrow Y \\
&\overset{(\star)}{\equiv} \forall Y.(\forall Z.(Z \Rightarrow 1+1) \Rightarrow (Z \Rightarrow 1+1) \Rightarrow Z \Rightarrow Y) \Rightarrow (1+1 \Rightarrow Y) \Rightarrow Y \\
&\overset{(***)}{\equiv} \forall Y.((\nu Z.(1+1) \times (1+1)) \Rightarrow Y) \Rightarrow (1+1 \Rightarrow Y) \Rightarrow Y \\
&\overset{(**)}{\equiv} \mu Y.(\nu Z.(1+1) \times (1+1)) + (1+1) \equiv 1+1+1+1+1+1+1
\end{aligned}$$

Figure 2 Short proof that a $\Lambda 2$-type has 6 inhabitants, using type isomorphisms.

Type Isomorphisms with the Yoneda Lemma. Our starting point is the observation that the *Yoneda lemma*, a cornerstone of category theory, is sometimes invoked [5, 16, 7, 35, 17] to justify some type isomorphisms in $\Lambda 2$ like e.g.

$$\forall X.(A \Rightarrow X) \Rightarrow (B \Rightarrow X) \equiv B \Rightarrow A \qquad \forall X.(X \Rightarrow A) \Rightarrow (X \Rightarrow B) \equiv A \Rightarrow B \qquad (\star)$$

which do not hold under $\beta\eta$-equivalence, but only under stronger equivalences. Such isomorphisms are usually justified by reference to the interpretation of polymorphic programs as *(di)natural transformations* [5], a well-known semantics of $\Lambda 2$ related to both parametricity [28] and free-theorems [36], and yielding a not yet well-understood equational theory over the programs of $\Lambda 2$ [15, 11, 23], that we call here the *ε-theory*. Other isomorphisms, like those in Fig. 1, can be justified in a similar way as soon as the language of $\Lambda 2$ is enriched with other type constructors like $1, 0, +, \times, \Rightarrow$ and *least/greatest fixpoints* $\mu X.A, \nu X.A$.

All such type isomorphisms have the effect of *eliminating* a quantifier, replacing it with a combination of monomorphic type constructors, and can be used to test if a polymorphic type has a *finite* number of inhabitants (as illustrated in Fig. 2) or, as suggested in [7], to devise *decidable tests* for program equivalence.

In this paper we develop a formal study of the elimination of quantifiers from polymorphic types using a class of type isomorphisms, that we will call *Yoneda type isomorphisms*, which generalize the examples above, and we explore its application for counting type inhabitants as well as to establish properties of program equivalence in $\Lambda 2$.

Eliminating Quantifiers with Yoneda Reduction. To give the reader a first glimpse of our approach, we compare the use of Yoneda type isomorphisms to count type inhabitants with some well-known *sufficient conditions* for a *simple* type A to have a unique or finitely many inhabitants [2, 9]: we show that whenever a simple type A satisfies either of these conditions, its universal closure $\forall \vec{X}.A$ can be converted (as in Fig. 2) to either 1 or $1+\cdots+1$ by applying Yoneda type isomorphisms and usual $\beta\eta$-isomorphisms.

We then turn to investigate the quantifier-eliminating rewriting over types arising from the left-to-right orientation of Yoneda type isomorphisms. A major obstacle here is that the rewriting must take into account possible applications of $\beta\eta$-isomorphisms, whose axiomatization is challenging in presence of the constructors $+, 0$ [13, 18] (as well as μ, ν). For this reason we introduce a family of rewrite rules, that we call *Yoneda reduction*, defined not directly over types but over a class of finite trees which represent the types of $\Lambda 2$ (but crucially *not* those made with $0, +, \dots$) up to $\beta\eta$-isomorphism.

Using this rewriting we establish some sufficient conditions for eliminating quantifiers, based on elementary graph-theoretic properties of such trees, which in turn provide some *new* sufficient conditions for the finiteness of type inhabitants of polymorphic types. First, we prove quantifier-elimination for the types satisfying a certain *coherence* condition which can be seen as an instance of the 2-SAT problem. We then introduce a more refined condition by associating each polymorphic type A with a value $\kappa(A) \in \{0, 1, \infty\}$, that we call the *characteristic of* A, so that whenever $\kappa(A) \neq \infty$, A rewrites into a monomorphic type, and when furthermore $\kappa(A) = 0$, A converges to a finite type. In the last case our method provides an effective way to count the inhabitants of A. The computation of $\kappa(A)$ is somehow reminiscent of linear logic *proof-nets*, as it is obtained by inspecting the existence of cyclic paths in a graph obtained by adding some "axiom-links" to the tree-representation of A.

Program Equivalence in System F with Finite Characteristic. Computing program equivalence under the ε-theory can be a challenging task, as this theory involves global permutations of rules which are difficult to detect and apply [11, 23, 34, 27, 36]. Things are even worse at the semantic level, since computing with dinatural transformations can be rather cumbersome, due to the well-known fact that such transformations need not compose [5, 21].

Nevertheless, our approach to quantifier-elimination based on the notion of characteristic provides a way to compute program equivalence *without* the appeal to ε-rules, free theorems and parametricity, since all polymorphic programs having types of finite characteristic can be embedded inside well-known monomorphic systems. To demonstrate this fact, we introduce two fragments $\Lambda 2^{\kappa \leqslant 0}$ and $\Lambda 2^{\kappa \leqslant 1}$ of $\Lambda 2$ in which types have a fixed finite characteristic, and we prove that these are equivalent, under the ε-theory, respectively, to the simply typed λ-calculus with finite products and co-products (or, equivalently, to the *free bicartesian closed category* $\mathbb{B}$), and to its extension with μ, ν-types (that is, to the *free cartesian closed* μ*-bicomplete category* $\mu\mathbb{B}$ [29, 6]). Using well-known facts about $\mathbb{B}$ and $\mu\mathbb{B}$ [31, 6, 22], we finally establish that the ε-theory is decidable in $\Lambda 2^{\kappa \leqslant 0}$ and undecidable in $\Lambda 2^{\kappa \leqslant 1}$.

Preliminaries and Notations

We will presuppose familiarity with the syntax of $\Lambda 2$ (in the version à la Church) and its extensions $\Lambda 2\mathsf{p}, \Lambda 2\mathsf{p}_{\mu\nu}$ with sum and product types, as well as μ and ν-types. We indicate by $\Lambda, \Lambda\mathsf{p}, \Lambda\mathsf{p}_{\mu\nu}$ their respective quantifier-free fragments. The syntax of these systems is recalled in App. A. We let $\mathsf{V} = \{X, Y, Z, \dots\}$ indicate the countable set of *type variables*.

Let S indicate any of the type systems above. We let $\Gamma \vdash_{\mathsf{S}} t : A$ indicate that the judgement $\Gamma \vdash t : A$ is derivable in S. We indicate as $t[x]$ a term with a unique free variable x, and we let $t[x] : A \vdash_{\mathsf{S}}^{\Gamma} B$ be shorthand for $\Gamma, x : A \vdash_{\mathsf{S}} t : B$.

A *theory* of S is a class of equations over well-typed terms satisfying usual congruence rules. Standard theories of $\Lambda 2, \Lambda 2\mathsf{p}, \Lambda 2\mathsf{p}_{\mu\nu}$ are those generated by $\beta\eta$-equivalence and by contextual equivalence, recalled in App. A. We will also consider a less standard theory, the ε-theory, described in App. B. For all theory T including $\beta\eta$-equivalence, we let $\mathbb{C}_{\mathsf{T}}(\mathsf{S})$ be the category whose objects are the types of S and whose arrows are the T-equivalence classes of terms $t[x] : A \vdash_{\mathsf{S}} B$. $\mathbb{C}_{\mathsf{T}}(\mathsf{S})$ is cartesian closed as soon as S contains products, meaning in particular that $\mathbb{C}_{\mathsf{T}}(\mathsf{S})(A \times B, C) \simeq \mathbb{C}_{\mathsf{T}}(\mathsf{S})(A, B \Rightarrow C)$.

By a T-*isomorphism*, indicated as $A \equiv_{\mathsf{T}} B$, we mean a pair of terms $t[x] : A \vdash_{\mathsf{S}} B$, $u[x] : B \vdash_{\mathsf{T}} A$ such that $t[u[x]] \simeq_{\mathsf{T}} x$ and $u[t[x]] \simeq_{\mathsf{T}} x$ (where $t[u[x]]$ is $t[x \mapsto u]$).

2 Yoneda Type Isomorphisms

In this section we introduce an axiomatization for a class of type isomorphisms that we call *Yoneda type isomorphisms*. For this we will rely on the well-known distinction between *positive* and *negative* occurrences of a variable X in a type A.

▶ **Notation 2.1.** *Throughout the text we indicate as* X *(resp.* X*) a positive (resp. negative) occurrence of* X*. When* B *occurs within a larger type* A*, we often note* B *as* $B\langle X\rangle$ *to indicate that all occurrences of the variable* X *in* B *are positive occurrences* in A*, or as* $B\langle X\rangle$ *to indicate that all occurrences of the variable* X *in* B *are negative occurrences* in A*. So for instance, when* B *only contains positive occurrences of* X*, we write the type* $A = X \Rightarrow B$ *as* $X \Rightarrow B\langle X\rangle$ *(since all positive occurrences of* X *in* B *are positive in* A*) and the type* $A' = B \Rightarrow X$ *as* $B\langle X\rangle \Rightarrow X$ *(since all positive occurrences of* X *in* B *are negative in* A*).*

The focus on positive/negative occurrences highlights a connection with to the so-called *functorial semantics* of $\Lambda 2$ [5, 15], in which types are interpreted as functors and typed programs as (di)natural transformations between such functors. More precisely, any positive type $A\langle X\rangle$ gives rise to a functor $\Phi^X_A : \mathbb{C}_{\mathsf{T}}(\mathsf{S}) \to \mathbb{C}_{\mathsf{T}}(\mathsf{S})$, any negative type $A\langle X\rangle$ gives rise to a functor $\Phi^X_A : \mathbb{C}_{\mathsf{T}}(\mathsf{S})^{\mathsf{op}} \to \mathbb{C}_{\mathsf{T}}(\mathsf{S})$ and, more generally, any type A gives rise to a functor $\Phi^X_A : \mathbb{C}_{\mathsf{T}}(\mathsf{S})^{\mathsf{op}} \times \mathbb{C}_{\mathsf{T}}(\mathsf{S}) \to \mathbb{C}_{\mathsf{T}}(\mathsf{S})$. In all such cases, the action of the functor on a type A is obtained by replacing positive/negative occurrences of X by A, and the action on programs can be defined inductively (we recall this construction in App. A, see also [11, 24]).

With types being interpreted as functors, a polymorphic term $t[x] : A \vdash_{\Lambda 2} B$ is interpreted as a transformation satisfying an appropriate naturality condition: when A and B have the same variance, $t[x]$ is interpreted as an ordinary natural transformation; instead, if A and B have mixed variances, then $t[x]$ is interpreted as a dinatural transformation.

Such (di)naturality conditions can be described syntactically through a class of equational rules over typed programs [11, 23] generating, along with the usual $\beta\eta$-equations, a theory of program equivalence that we call the *ε-theory*.[1] These equational rules are usually interpreted as parametricity conditions [28], or as instances of *free theorems* [36].

As mentioned in the introduction, our goal here is not that of investigating the ε-theory directly, but rather to explore a class of type isomorphisms that hold under this theory (that is, of isomorphisms in the syntactic categories $\mathbb{C}_{\varepsilon}(\mathsf{S})$, with $\mathsf{S} = \Lambda 2, \Lambda 2\mathsf{p}, \Lambda 2\mathsf{p}_{\mu\nu}$). For example, in functorial semantics a type of the form $\forall X.A\langle X\rangle \Rightarrow B\langle X\rangle$ is interpreted as the set of natural transformations between the functors $A\langle X\rangle$ and $B\langle X\rangle$. Now, if $A\langle X\rangle$ is of the form $A_0 \Rightarrow X$ (i.e. it is a *representable* functor), using the ε-theory we can deduce (see App. B) a "Yoneda lemma" in the form of the quantifier-eliminating isomorphism below:

$$\forall X.(A_0 \Rightarrow X) \Rightarrow B\langle X\rangle \;\equiv\; B\langle X \mapsto A_0\rangle \tag{1}$$

Similarly, if $A\langle X\rangle, B\langle X\rangle$ are both negative and $A\langle X\rangle$ is of the form $X \Rightarrow A_0$ (i.e. it is a *co-representable* functor), we can deduce another quantifier-eliminating isomorphism:

$$\forall X.(X \Rightarrow A_0) \Rightarrow B\langle X\rangle \;\equiv\; B\langle X \mapsto A_0\rangle \tag{2}$$

Observe that both isomorphisms ($\star$) from the Introduction are instances of (1) or (2).

[1] We define this theory formally in App. B, but this is not necessary to understand this paper.

As we admit more type-constructors in the language, we can use the ε-theory to deduce stronger schemas for eliminating quantifiers. For instance, using *least* and *greatest fixed points* $\mu X.A\langle X\rangle, \nu X.A\langle X\rangle$ of positive types, we can deduce the stronger schemas [35] below.

$$\forall X.(A\langle X\rangle \Rightarrow X) \Rightarrow B\langle X\rangle \equiv B\langle X \mapsto \mu X.A\langle X\rangle\rangle \tag{3}$$
$$\forall X.(X \Rightarrow A\langle X\rangle) \Rightarrow C\langle X\rangle \equiv C\langle X \mapsto \nu X.A\langle X\rangle\rangle \tag{4}$$

Note that (1) and (2) can be deduced from (3) and (4) using the isomorphisms $\mu X.A \equiv_{\beta\eta} \nu X.A \equiv_{\beta\eta} A$, when X does not occur in A. Moreover, adding sum and product types enables the elimination of the quantifier $\forall X$ also from a type of the form $A = \forall X.(A_{11}\langle X\rangle \Rightarrow A_{12}\langle X\rangle \Rightarrow X) \Rightarrow (A_{21}\langle X\rangle \Rightarrow A_{22}\langle X\rangle \Rightarrow X) \Rightarrow B\langle X\rangle$ by using $\beta\eta$-isomorphisms:

$$A \equiv_{\beta\eta} \forall X.\left(\left(\textstyle\sum_{i=1,2}\prod_{j=1,2} A_{ij}\langle X\rangle\right) \Rightarrow X\right) \Rightarrow B\langle X\rangle \equiv B\left\langle X \mapsto \mu X.\left(\textstyle\sum_{i=1,2}\prod_{j=1,2} A_{ij}\langle X\rangle\right)\right\rangle$$

These considerations lead to introduce the following class of isomorphisms:

▶ **Definition 1.** *A* Yoneda type isomorphism *is any instance of the schemas* $\equiv_X$, $\equiv_X$ *below*

$$\forall X.\forall\vec{Y}.\left\langle \forall\vec{Z}_k.\langle A_{jk}\langle X\rangle\rangle_j \Rightarrow X\right\rangle_k \Rightarrow B\langle X\rangle \quad \equiv_X \quad \forall\vec{Y}.B\left\langle X \mapsto \{\mu X.\}\textstyle\sum_k\left(\exists\vec{Z}_k.\prod_j A_{jk}\langle X\rangle\right)\right\rangle$$

$$\forall X.\forall\vec{Y}.\left\langle \forall\vec{Z}_k.X \Rightarrow A_j\langle X\rangle\right\rangle_k \Rightarrow B\langle X\rangle \quad \equiv_X \quad \forall\vec{Y}.B\left\langle X \mapsto \{\nu X.\}\forall\vec{Z}_k.\prod_j A_j\langle X\rangle\right\rangle$$

where, given a list $L = \langle i_1,\dots,i_k\rangle$*, and an* L*-indexed list of types* $(A_i)_{i\in L}$*,* $\langle A_i\rangle_{i\in L} \Rightarrow B$ *is a shorthand for* $A_{i_1} \Rightarrow \dots \Rightarrow A_{i_k} \Rightarrow B$*, the expressions* $\{\mu X.\}$*,* $\{\nu X.\}$ *indicate that the binder* $\mu X.$ *(resp.* $\nu X.$*) is applied only if* X *(resp.* X*) actually occurs in some of the* A_{jk} *(resp.* A_j*), and* $\exists\vec{Y}.A$ *is a shorthand for* $\forall Y'.(\forall\vec{Y}.A \Rightarrow Y') \Rightarrow Y'$*).*

For all types A, B *of* $\mathsf{\Lambda 2p}_{\mu\nu}$*, we write* $A \equiv_{\mathsf{Y}} B$ *when* A *can be converted to* B *using* $\equiv_X, \equiv_X$ *and the partial*[2] *axiomatization of* $\beta\eta$*-isomorphisms in Fig. 9-11 (App. A).*

Since $\equiv_X$ and $\equiv_X$ are ε-isomorphisms (see App. B), whenever $A \equiv_{\mathsf{Y}} B$, A and B are interpreted as isomorphic objects in all dinatural and parametric models of $\mathsf{\Lambda 2}$.

3 Counting Type Inhabitants with Yoneda Type Isomorphisms

A first natural application of Yoneda type isomorphisms is to *count* the inhabitants of a *simple* type: given such a type A, with free variables $\vec{X}$, if $\forall\vec{X}.A \equiv_{\mathsf{Y}} 0 + 1 + \dots + 1 = \sum_{i=1}^{k} 1$, then A has exactly k proofs (up to ε-equivalence). Let us start with a "warm-up" example.

▶ **Example 2.** In [9] it is proved that the type $A = ((((X \Rightarrow Y) \Rightarrow X \Rightarrow Z) \Rightarrow (Y \Rightarrow Z) \Rightarrow W) \Rightarrow (Y \Rightarrow Z) \Rightarrow W$ has a unique inhabitant. Here's a quick proof of $\forall XYZW.A \equiv_{\mathsf{Y}} 1$:

$$\begin{aligned}
&\forall XYZW.((((X \Rightarrow Y) \Rightarrow X \Rightarrow Z) \Rightarrow (Y \Rightarrow Z) \Rightarrow W) \Rightarrow (Y \Rightarrow Z) \Rightarrow W\\
&\equiv_W \forall XYZ.(Y \Rightarrow Z) \Rightarrow \Big(((X \Rightarrow Y) \Rightarrow X \Rightarrow Z) \times (Y \Rightarrow Z)\Big)\\
&\equiv_Z \forall XY.\big((X \Rightarrow Y) \Rightarrow X \Rightarrow Y\big) \times \big(Y \Rightarrow Y\big)\\
&\equiv_{\beta\eta} \Big(\forall XY.\big((X \Rightarrow Y) \Rightarrow X \Rightarrow Y\big)\Big) \times \Big(\forall Y.Y \Rightarrow Y\Big)\\
&\equiv_X \Big(\forall Y.Y \Rightarrow Y\Big) \times \Big(\forall Y.Y \Rightarrow Y\Big) \equiv_Y 1 \times 1 \equiv_{\beta\eta} 1
\end{aligned}$$

[2] The axiomatization in Fig. 9-11 is complete for the $\beta\eta$-isomorphisms of $\mathsf{\Lambda 2}$ [12], but fails to be complete (already at the propositional level) in presence of sums and the empty type [13, 18].

The literature on counting simple type inhabitants is vast (e.g. [2, 9, 8, 32]) and includes both complete algorithms and simpler *sufficient conditions* for a given type to have a unique or finite number of inhabitants. The latter provide then an ideal starting point to test our axiomatic theory of type isomorphisms, as several of these conditions are based on properties like the number of positive/negative occurrences of variables.

We tested Yoneda type isomorphisms on two well-known sufficient conditions for unique inhabitation. A simple type A is *balanced* when any variable occurring free in A occurs exactly once as X and exactly once as X. A is *negatively non-duplicated* if no variable occurs twice in A as X. An inhabited simple type which is balanced or negatively duplicated has exactly one inhabitant [2]. The theory $\equiv_{\mathsf{Y}}$ subsumes these conditions in the following sense:

▶ **Proposition 3.** *Let $A[X_1, \dots, X_n]$ be an inhabited simple type with free variables $X_1, \dots, X_n$. If $A[X_1, \dots, X_n]$ is either balanced or negatively non-duplicated, then $\forall X_1 \dots \forall X_n.A \equiv_{\mathsf{Y}} 1$.*

Observe that, since the type in Example 2 is neither balanced nor negatively non-duplicated, type isomorphisms provide a stronger condition than the two above.

We tested another well-known property, dual to one of the previous ones: a simple type A is *positively non-duplicated* if no variable occurs twice in A as X. A positively non-duplicated simple type has a finite number of proofs [9]. We reproved this fact using type isomorphisms, but this time only in a restricted case. Let the *depth* $d(A)$ of a simple type A be defined by $d(X) = 0$, $d(A \Rightarrow B) = \max\{d(A) + 1, d(B)\}$.

▶ **Proposition 4.** *Let $A[X_1, \dots, X_n]$ be an inhabited simple type with free variables $X_1, \dots, X_n$. If A is positively non-duplicated and $d(A) \leqslant 2$, then $\forall X_1 \dots \forall X_n.A \equiv_{\mathsf{Y}} 0 + 1 + \dots + 1$.*

4 From Polymorphic Types to Polynomial Trees

Read from left to right, the schemas $\equiv_X, \equiv_X$ yield rewriting rules over $\Lambda 2\mathsf{p}_{\mu\nu}$-types which *eliminate* occurrences of polymorphic quantifiers. Yet, a major obstacle to study this rewriting is that the application of $\equiv_X, \equiv_X$ might depend on the former application of $\beta\eta$-isomorphisms (as we did for instance in the previous section). Already for the propositional fragment $\Lambda\mathsf{p}$, the $\beta\eta$-isomorphisms are not finitely axiomatizable and it is not yet clear if a decision algorithm exists at all (see [13, 18]). This implies in particular that a *complete* criterion for the conversion of a $\Lambda 2$-type to a monomorphic (or even finite) type can hardly be computable.

For this reason, we restrict our goal to establishing some efficiently recognizable (in fact, polytime) *sufficient conditions* for quantifier-elimination. Moreover, we will exploit the well-known fact that the constructors $0, 1, +, \times, \mu, \nu$ can be encoded inside $\Lambda 2$ to describe our rewriting entirely within (a suitable representation of) $\Lambda 2$-types, for which $\beta\eta$-isomorphisms are completely axiomatized by the rules in Fig. 9 (see [12]).

Even if one restricts to $\Lambda 2$-types, recognizing if one of the schemas $\equiv_X, \equiv_X$ applies to a $\Lambda 2$-type $\forall X.A$ might still require to first apply some $\beta\eta$-isomorphisms. For example, consider the $\Lambda 2$-type $A = \forall X.((Y \Rightarrow X) \Rightarrow Y) \Rightarrow (Y \Rightarrow X) \Rightarrow Y$. In order to eliminate the quantifier $\forall X$ using $\equiv_X$, we first need to apply the $\beta\eta$-isomorphism $A \Rightarrow (B \Rightarrow C) \equiv_{\beta\eta} B \Rightarrow (A \Rightarrow C)$, turning A into $\forall X.(Y \Rightarrow X) \Rightarrow ((Y \Rightarrow X) \Rightarrow Y) \Rightarrow Y$, which is now of the form $\forall X.(B\langle X\rangle \Rightarrow X) \Rightarrow C\langle X\rangle$, with $B\langle X\rangle = Y$ and $C\langle X\rangle = ((Y \Rightarrow X) \Rightarrow Y) \Rightarrow Y$. We can then apply $\equiv_X$, yielding $((Y \Rightarrow Y) \Rightarrow Y) \Rightarrow Y$.

To obviate this problem, we introduce below a representation of $\Lambda 2$-types as labeled trees so that $\beta\eta$-isomorphic types are represented by the same tree. In the next section we will reformulate the schemas $\equiv_X, \equiv_X$ as reduction rules over such trees. This approach drastically simplifies the study of this rewriting, and will allow us to establish conditions for quantifier-elimination based on elementary graph-theoretic properties.

A $\beta\eta$-invariant representation of $\Lambda 2$-types. We introduce a representations of $\Lambda 2$-types as rooted trees whose leaves are labeled by *colored* variables, with colors being any $c \in \mathsf{Colors} = \{\mathrm{blue}, \mathrm{red}\}$. We indicate such variables as either X^c or simply as X, X. Moreover, we indicate as $\bar{c}$ the unique color different from c.

By a rooted tree we indicate a finite connected acyclic graph with a chosen vertex, called its root. If $(\mathscr{G}_i)_{i\in I}$ is a finite family of rooted trees, we indicate as $\begin{array}{c}\{\mathscr{G}_i\}_{i\in I}\\ |\\ u\end{array}$ the tree with root u obtained by adding an edge from any of the roots of the trees $\mathscr{G}_i$ to u.

▶ **Definition 5.** *The sets $\mathcal{E}$ and $\mathcal{E}$ of* positive *and* negative $\Lambda 2$-trees *are inductively defined by:*

$$\mathsf{E} \quad := \quad \begin{array}{ccc}\{\mathsf{E}_i\}_{i\in I} & & X\\ & \diagdown\ \diagup & \\ & \vec{Y} & \end{array} \qquad\qquad \mathsf{E} \quad := \quad \begin{array}{ccc}\{\mathsf{E}_i\}_{i\in I} & & X\\ & \diagdown\ \diagup & \\ & \vec{Y} & \end{array}$$

where X is a variable, $\vec{Y}$ indicates a finite set of variables, and the edge in E (resp. E) with label X (resp. X) is called the head *of E (resp. of E). The trees* $\begin{array}{ccc}\{\,\}_\varnothing & & X\\ & \diagdown\ \diagup & \\ & \varnothing & \end{array}$ *and* $\begin{array}{ccc}\{\,\}_\varnothing & & X\\ & \diagdown\ \diagup & \\ & \varnothing & \end{array}$ *are indicated simply as X and X.*

Free and *bound* variables of a tree $\mathsf{E} = \begin{array}{ccc}\{\mathsf{E}_i\}_{i\in I} & & X\\ & \diagdown\ \diagup & \\ & \vec{Y} & \end{array}$ are defined by $\mathsf{f}\mathcal{V}(\mathsf{E}) = \bigcup_{i=1}^n \mathsf{f}\mathcal{V}(\mathsf{E}_i) \cup \{X\} - \vec{Y}$ and $\mathsf{b}\mathcal{V}(\mathsf{E}) = \bigcup_{i=1}^n \mathsf{b}\mathcal{V}(\mathsf{E}_i) \cup \vec{Y}$.

We can associate a positive and a negative $\Lambda 2$-tree to any $\Lambda 2$-type as follows. Let us say that a type A of $\Lambda 2$ is *in normal form* (shortly, *in NF*) if $A = \forall\vec{Y}.A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow X$ where each of the variables in $\vec{Y}$ occurs in $A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow X$ at least once, and the types $A_1, \ldots, A_n$ are in normal form. It can be checked that any $\Lambda 2$-type is $\beta\eta$-isomorphic to a type in NF, that we indicate as $\mathsf{NF}(A)$.

▶ **Definition 6.** *For all $A \in \Lambda 2$, with $\mathsf{NF}(A) = \forall\vec{Y}.A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow X$, let*

$$\mathsf{t}(A) = \begin{array}{ccc}\{\mathsf{t}(A_i)\}_{i=1,\ldots,n} & & X\\ & \diagdown\ \diagup & \\ & \vec{Y} & \end{array} \qquad\qquad \mathsf{t}(A) = \begin{array}{ccc}\{\mathsf{t}(A_i)\}_{i=1,\ldots,n} & & X\\ & \diagdown\ \diagup & \\ & \vec{Y} & \end{array}$$

The tree-representation of $\Lambda 2$-types captures $\beta\eta$-isomorphism classes, in the sense that $A \equiv_{\beta\eta} B$ iff $\mathsf{t}(A) = \mathsf{t}(B)$ (this is proved in detail in [26]). For instance, the two $\beta\eta$-isomorphic types $\forall XY.(X \Rightarrow X) \Rightarrow (\forall Z.X \Rightarrow Z) \Rightarrow (Y \Rightarrow X) \Rightarrow Y$ and $\forall X.(X \Rightarrow X) \Rightarrow \forall Y.(Y \Rightarrow X) \Rightarrow (X \Rightarrow \forall Z.Z) \Rightarrow Y$ translate into the same $\Lambda 2$-tree, shown in Fig. 4b (where underlined node labels and dashed edges can be ignored, for now).

Polynomial Trees. To formulate the schemas $\equiv_X, \equiv_X$ in the language of rooted trees we exploit an encoding of the types of the form $\mu X.\exists\vec{Y}.\sum_{k\in K}\prod_{j\in J_k} A_{jk}\langle X\rangle$ and $\nu X.\forall\vec{Y}.\prod_{j\in J} A_j\langle X\rangle$ as certain special trees employing two new constants $\bullet$ and $\blacktriangle$. This encoding is easily seen to be a small variant, in the language of finite trees, of the usual second-order encodings.

We first introduce a handy notation for "polynomial" types, i.e. types corresponding to a generalized sum of generalized products. Following [14], any such type A is completely determined by a diagram of finite sets $I \xleftarrow{f} J \xrightarrow{g} K$ and a I-indexed family of types $(A_i)_{i\in I}$, so that $A = \sum_{k\in K}\prod_{j\in J_k} A_{f(j)}$, where $J_k := g^{-1}(k)$. In the following, we will call the given of a finite diagram $I \xleftarrow{f} J \xrightarrow{g} K$, and an I-indexed family $(a_i)_{i\in I}$ a *polynomial family*, and indicate it simply as $(a_{jk})_{k\in K, j\in J_k}$ (in fact, we already implicitly used this notation in Def. 1).

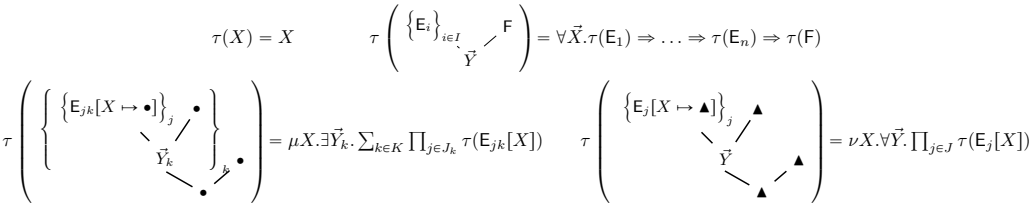
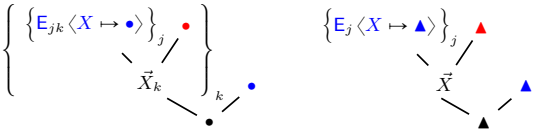


Figure 3 Translation of simple polynomial trees into monomorphic types.

We now enrich the class of Λ2-trees as follows:

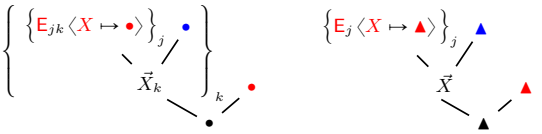
▶ **Definition 7** (Polynomial trees). *Let* $\bullet, \blacktriangle$ *indicate two new constants.*

- *the set* $\mathcal{P}$ *of* positive polynomial trees *is defined by adding to the clauses defining positive* Λ2*-trees two new clauses:*



  where X *is some variable,* $(\mathsf{E}_{jk}\langle X\rangle)_{k,j}$ *(resp.* $(\mathsf{E}_j\langle X\rangle)_j$*) is a polynomial family (resp. a family) of positive polynomial trees with no occurrence of* X*, and* $(\vec{X}_k)_{k\in K}$ *is a* K*-indexed family of finite sets of variables.*

- *the set* $\mathcal{P}$ *of* negative polynomial trees *is defined by adding to the clauses defining negative* Λ2*-trees two new clauses:*



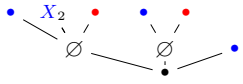
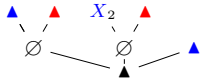
  where X *is some variable,* $(\mathsf{E}_{jk}\langle X\rangle)_{k,j}$ *(resp.* $(\mathsf{E}_j\langle X\rangle)_j$*) is a polynomial family (resp. a family) of negative polynomial trees with no occurrence of* X*, and* $(\vec{X}_k)_{k\in K}$ *is a* K*-indexed family of finite sets of variables.*

We indicate by $\mathcal{P}$ *the set of all polynomial trees, and by* $\mathcal{P}_0$ *the set of all polynomial trees with no bound variables, which are called* simple*.*

Any polynomial tree $\mathsf{E} \in \mathcal{P}$ can be converted into a type $\tau(\mathsf{E})$ of $\Lambda 2\mathsf{p}_{\mu\nu}$ as illustrated in Fig. 3. It is easily checked that, whenever E is simple, $\tau(\mathsf{E})$ has no quantifier. Moreover, one can check that for all Λ2-type, $\tau(\mathsf{t}(A)) = A$.

We conclude this section with some basic example of polynomial trees.

▶ **Example 8.**

- The constant types 0 and 1 are represented as positive/negative trees by $\mathbf{0} =$  , $\mathbf{0} =$  and $\mathbf{1} =$  , $\mathbf{1} =$  .
- The diagram $\{1,2\} \overset{1,3\mapsto 1;2\mapsto 2}{\leftarrow} \{1,2,3\} \overset{1,2\mapsto 1;3\mapsto 2}{\rightarrow} \{1,2\}$, along with the family $(X_i)_{i\in\{1,2\}}$, yields the polynomial family $(\mathsf{E}_{jk})_{k,j}$, with $\mathsf{E}_{11} = \mathsf{E}_{32} = X_1$ and $\mathsf{E}_{21} = X_2$, and yields the polynomial tree  encoding the type $\mu X_1.(X_1 \times X_2) + X_1$.
- The diagram $\{1,2\} \overset{id}{\leftarrow} \{1,2\} \rightarrow \{1\}$ with the same family as above yields the polynomial tree  encoding the type $\nu X_1.X_1 \times X_2$.

(a) $\forall X.(X \Rightarrow X) \Rightarrow (\forall Y.X \Rightarrow Y) \Rightarrow X$. **(b)** $\forall XY.(X \Rightarrow X) \Rightarrow (\forall Z.X \Rightarrow Z) \Rightarrow (Y \Rightarrow X) \Rightarrow Y$.

Figure 4 Polynomial trees with highlighted modular nodes and modular pairs.

5 Yoneda Reduction

In this section we introduce a family of rewriting rules $\rightsquigarrow_X, \rightsquigarrow_X$ over polynomial trees, that we call *Yoneda reduction*, which correspond to the left-to-right orientation of the isomorphisms $\equiv_X, \equiv_X$. We will adopt the following conventions:

▶ **Notation 5.1.** *We make the assumption that all bound variables of a polynomial tree* E *are distinct. More precisely, for any* $X \in \mathsf{bV}(\mathsf{E})$*, we suppose there exist unique nodes* r_X *and* h_X *such that* $\mathsf{r}_X : \vec{X}$*, for some set of variable* $\vec{X}$ *such that* $X \in \vec{X}$*, and* h_X *is the head of the sub-tree whose root is* r_X*.*

We will call two distinct nodes *parallel* if they are immediate successors of the same node, and we let the distance $d(\alpha, \beta)$ between two nodes in a polynomial tree be the number of edges of the unique path from α to β.

Using polynomial trees we can identify when a quantifier can be eliminated from a type independently from $\beta\eta$-isomorphisms, by inspecting a simple condition on the tree-representation of the type based on the notion of *modular* node, introduced below.

▶ **Definition 9.** *For all* $X \in \mathsf{bV}(\mathsf{E})$*, a terminal node* $\alpha : X^c$ *in* E *is said* modular *if* $\alpha \neq \mathsf{h}_X$*,* $1 \leqslant d(\alpha, \mathsf{r}_X) \leqslant 2$ *and* α *has no parallel node of label* X^c*. A pair of nodes of the form* $(\alpha : X, \beta : X)$ *is called a* X-pair*, and a* X*-pair is said* modular *if one of its nodes is modular.*

In the trees in Fig. 4 the modular nodes are underlined and the modular pairs are indicated as dashed edges.

▶ **Definition 10.** *A variable* $X \in \mathsf{bV}(\mathsf{E})$ *is said* eliminable *when every* X*-pair of* E *is modular. For every color* c*, we furthermore call* X c-eliminable *if every node* $\alpha : X^{\overline{c}}$ *is modular.*

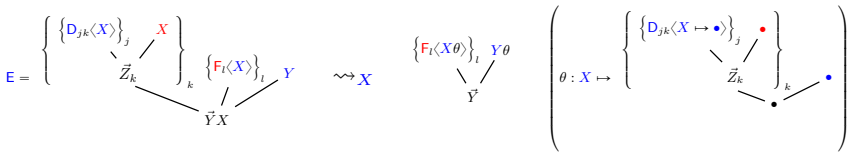
We let eliminable be a shorthand for "blue-eliminable" and eliminable be a shorthand for "red-eliminable". These notions are related as follows:

▶ **Lemma 11.** *X is eliminable iff it is either* eliminable *or* eliminable*.*

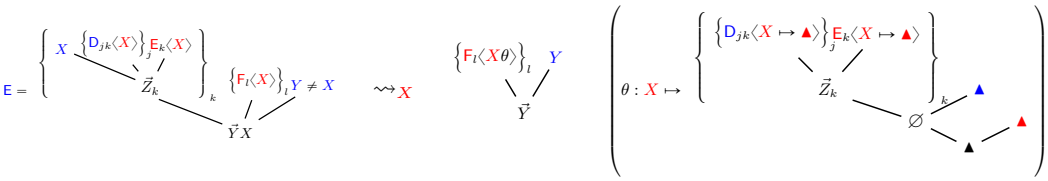
Proof. If X is neither eliminable nor eliminable, then there exist non-modular nodes $\alpha : X, \beta : X$, whence the X-pair (α, β) is not modular. Conversely, suppose X is eliminable but not eliminable. Hence there is a non modular node $\alpha : X$. For all node $\beta : X$, since the X-pair (α, β) is modular, β is modular. We deduce that X is eliminable. ◀

▶ **Example 12.** The variable X is eliminable but not eliminable in the tree in Fig. 4a, and it is both eliminable and eliminable in the tree in Fig. 4b.

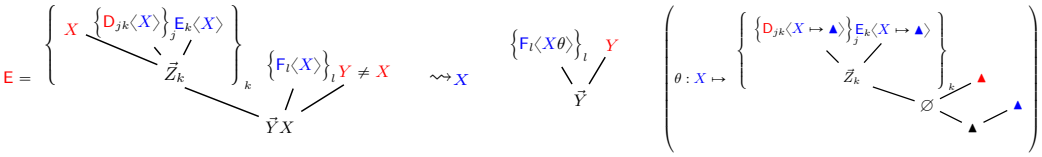
The proposition below shows that a variable X is eliminable (resp. eliminable) in the tree of a $\Lambda 2$-type $\forall X.A$ exactly when $\forall X.A$ matches, up to $\beta\eta$-isomorphisms, with the left-hand type of the schema $\equiv_X$ (resp. $\equiv_X$).



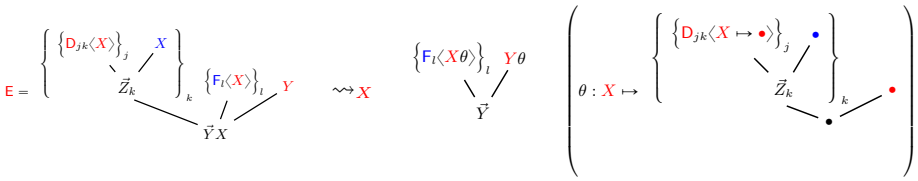
(a) X eliminable in E.



(b) X eliminable in E.



(c) X eliminable in E.



(d) X eliminable in E.

Figure 5 Yoneda reduction of X-eliminable trees.

▶ **Proposition 13.** *X is eliminable in E iff E is as in Fig. 5a left, for some polynomial family $(\mathsf{D}_{jk}\langle X\rangle)_{k\in K, j\in J_k}$, family $(\mathsf{F}_l\langle X\rangle)_{l\in L}$ and blue variable Y (possibly X itself).*

- *X is eliminable in E iff E is as in Fig. 5b left, for some polynomial family $(\mathsf{D}_{jk}\langle X\rangle)_{k\in K, j\in J_k}$, family $(\mathsf{E}_k\langle X\rangle)_{k\in K}$, family $(\mathsf{F}_l\langle X\rangle)_{l\in L}$ and blue variable $Y \neq X$.*
- *X is eliminable in E iff E is as in Fig. 5c left, for some polynomial family $(\mathsf{D}_{jk}\langle X\rangle)_{k\in K, j\in J_k}$, family $(\mathsf{E}_k\langle X\rangle)_{k\in K}$, family $(\mathsf{F}_l\langle X\rangle)_{l\in L}$ and red variable $Y \neq X$.*
- *X is eliminable in E iff E is as in Fig. 5d left, for some polynomial family $(\mathsf{D}_{jk}\langle X\rangle)_{k\in K, j\in J_k}$, family $(\mathsf{F}_l\langle X\rangle)_{l\in L}$ and red variable Y (possibly X itself).*

For all four cases of Prop. 13 we define a rewriting rule which eliminates X.

▶ **Definition 14** (Yoneda reduction). *Let $\mathsf{F} \in \mathcal{P}$ and $X \in \mathsf{b}\mathcal{V}(\mathsf{F})$ be an eliminable variable. The rules $\mathsf{F} \rightsquigarrow_{X^c} \mathsf{F}'$ consist in replacing the subtree E of F rooted in r_X as illustrated in Fig. 5.*

▶ **Example 15.** The tree in Fig. 4b rewrites as illustrated in Fig. 6.

By inspecting the rules in Fig. 5 one can check that $\mathsf{E} \rightsquigarrow_{X^c} \mathsf{E}'$ implies $\tau(\mathsf{E}) \equiv_{\mathsf{Y}} \tau(\mathsf{E}')$. From this we can deduce by induction:

▶ **Lemma 16.** *For all $\Lambda 2$-type A, if $\mathsf{t}(A) \rightsquigarrow^* \mathsf{E} \in \mathcal{P}_0$, then $A \equiv_{\mathsf{Y}} \tau(\mathsf{E}) \in \Lambda\mathsf{p}_{\mu\nu}$.*

The lemma above suggests to study the elimination of quantifiers from $\Lambda 2$-types by studying the convergence of $\Lambda 2$-trees onto simple polynomial trees. This will be our next goal.

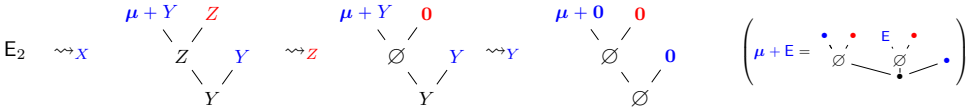


Figure 6 Yoneda reduction of the polynomial tree E_2 from Fig. 4b.

6 The Characteristic of a Polymorphic Type

In this section we exploit Yoneda reduction to establish two sufficient conditions to convert a $\Lambda 2$-type A into a quantifier-free type A' such that $A \equiv_{\mathsf{Y}} A'$.

The Coherence Condition. When a reduction is applied to E, several sub-trees of E can be either erased or copied and moved elsewhere. Hence, the resulting tree E' might well have a greater size and even a larger number of bound variables than E. Nevertheless, sequences of Yoneda reductions always terminate, as one can define a measure which decreases at each step (see [26]).

▶ **Proposition 17.** *There is no infinite sequence of Yoneda reductions.*

Although sequences of reductions always terminate, they need not terminate on a simple polynomial tree, that is, on the encoding of a monomorphic type. This can be due to several reasons. Firstly, one bound variable might not be eliminable. Secondly, even if all variables are eliminable, this property need not be preserved by reduction. For example, take the type $A = \forall X.\forall Y.(X \Rightarrow Y \Rightarrow X) \Rightarrow ((Y \Rightarrow X) \Rightarrow W) \Rightarrow Z$: although X and Y are both eliminable (in the associated tree), if we apply a reduction to X, then Y ceases to be eliminable, and similarly if we reduce Y first. Such conflicts can be controlled by imposing a suitable *coherence* relation on variables.

▶ **Definition 18.** *Let* E *be a polynomial tree,* $X, Y \in \mathsf{bV}(\mathsf{E})$ *and* $c, d \in \mathsf{Colors}$. X^c *and* Y^d *are said* coherent *if there exists no parallel modular nodes of the form* $\alpha : X^{\bar{c}}, \beta : Y^{\bar{d}}$ *in* E.

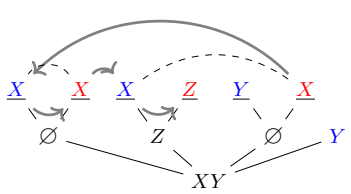
▶ **Example 19.** In the tree in Fig. 4b, Y and Z are coherent, while X and Y are not.

▶ **Definition 20** (coherence condition)**.** *Let* E *be a polynomial tree. A* valuation of E *is any map* $\phi : \mathsf{bV}(\mathsf{E}) \to \mathsf{Colors}$. *For all valuation* ϕ *of* E*, we call* E ϕ-coherent *if for all* $X \in \mathsf{bV}(\mathsf{E})$*,* X *is* $\phi(X)$*-eliminable, and moreover for all* $Y \neq X \in \mathsf{bV}(\mathsf{E})$*,* $X^{\phi(X)}$ *is coherent with* $Y^{\phi(Y)}$*. We call* E coherent *if it is* ϕ*-coherent for some valuation* ϕ *of* E.

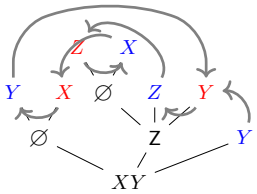
▶ Remark 21 (Coherence is an instance of 2-SAT). The problem of checking if a polynomial tree E is coherent can be formulated as an instance of 2-SAT (a well-known polytime problem): consider n Boolean variables $x_1, \ldots, x_n$ (one for each bound variable of E), and let a_i^c be x_i if $c = \text{blue}$ and $\neg x_i$ if $c = \text{red}$. Consider then the 2-CNF $A \wedge B$, where A is the conjunction of all $a_i^c \vee a_i^c$ such that X_i is not $\bar{c}$-eliminable in E, and B is the conjunction of all $a_i^c \vee a_j^d$, for all incoherent $X_i^{\bar{c}}$ and $X_j^{\bar{d}}$. Then a coherent valuation of E is the same as a model of $A \wedge B$.

As observed before, a reduction $\mathsf{E} \rightsquigarrow_X \mathsf{E}'$ might copy or erase some bound variables of E. One can then define a map $g : \mathsf{bV}(\mathsf{E}') \to \mathsf{bV}(\mathsf{E})$ associating any variable in E' with the corresponding variable in E of which it is a copy. A sequence of reductions $\mathsf{E}_0 \rightsquigarrow_{X_1^{c_1}} \cdots \rightsquigarrow_{X_n^{c_n}} \mathsf{E}_n$ induces then, for $1 \leqslant i \leqslant n$, maps $g_i : \mathsf{bV}(\mathsf{E}_i) \to \mathsf{bV}(\mathsf{E}_{i-1})$, and we let $G_i : \mathsf{bV}(\mathsf{E}_i) \to \mathsf{bV}(\mathsf{E}_0)$ be $g_1 \circ g_2 \circ \cdots \circ g_i$.

The rewriting properties of coherent trees are captured by the following notion:



(a) Alternate path in $\forall XY.(X \Rightarrow X) \Rightarrow (\forall Z.X \Rightarrow Z) \Rightarrow (Y \Rightarrow X) \Rightarrow Y$.



(b) Cyclic alternate path in $\forall XY.(Y \Rightarrow X) \Rightarrow (\forall Z.(Z \Rightarrow X) \Rightarrow (Z \Rightarrow Y)) \Rightarrow Y$.

Figure 7 Examples of alternate paths.

▶ **Definition 22** (standard reduction). *A sequence of reductions* $\mathsf{E}_0 \rightsquigarrow_{X_1^{c_1}} \cdots \rightsquigarrow_{X_n^{c_n}} \mathsf{E}_n$ *is said* standard *if for all* $i, j = 1, \ldots, n$, $G_i(X_i)^{c_i}$ *is coherent with* $G_j(X_j)^{c_j}$ *in* E_0. *We say that* E strongly converges under standard reduction *if all standard reductions starting from* E *terminate on a simple polynomial tree.*

Using the fact that coherence is stable under standard reduction (see [26]) we obtain:

▶ **Theorem 23.** E *is coherent iff* E *strongly converges under standard reduction.*

Using Lemma 16 we further deduce:

▶ **Corollary 24.** *Let* A *be a type of* $\Lambda 2$. *If* $\mathsf{t}(A)$ *is coherent, then there exists a* $\Lambda\mathsf{p}_{\mu\nu}$*-type* A' *such that* $A \equiv_{\mathsf{Y}} A'$.

The Characteristic. We now introduce a refined condition for coherent trees, which can be used to predict whether a type rewrites into a finite type (i.e. one made up from $0, 1, +, \times, \Rightarrow$ only) or into one using μ, ν-constructors.

An intuition from Section 2 is that, for a type of the form $\forall X.(A\langle X\rangle \Rightarrow X) \Rightarrow B\langle X\rangle$ (which rewrites into $B\langle X \mapsto \mu X.A\langle X\rangle\rangle$) to reduce to one without μ, ν-types, the variable X *must not occur in* A at all. However, the property "X does not occur in A" need not be preserved under reduction. Instead, we will define a stronger condition that is preserved by standard reduction by inspecting a class of *paths* in the tree of a type.

▶ **Definition 25.** *Let* E *be a polynomial* $\Lambda 2$*-tree and let* $\preceq$ *indicate the natural order on the nodes of* E *having the root of* E *as its minimum. A* down-move *in* E *is a pair* $\alpha \rhd \beta$*, such that, for some bound variable* $X \in \mathsf{bV}(\mathsf{E})$, (α, β) *is an* X*-pair and* β *is modular. An* up-move *in* E *is a pair* $\alpha \rhd \beta$ *such that* $\alpha \neq \beta$, α *is a modular node with immediate predecessor* γ, $\beta : X$ *for some* $X \in \mathsf{bV}(\mathsf{E})$, *and* $\gamma \preceq \beta$. *An* alternating path *in* E *is a sequence of nodes* $\alpha_0 \ldots \alpha_{2n}$ *such that* $\alpha_{2i} \rhd \alpha_{2i+1}$ *is a down-move and* $\alpha_{2i+1} \rhd \alpha_{2i+2}$ *is an up-move.*

In Fig. 7b are illustrated some alternating paths. Observe that whenever X occurs in $A\langle X\rangle$, we can construct a *cyclic* alternate path in the tree of $\forall X.(A\langle X\rangle \Rightarrow X) \Rightarrow B\langle X\rangle$: down-move from an occurrence of X in A to the modular node labeled X, then up-move back to X. We deduce that if no cyclic alternate path exists, then any subtype of the form above must be such that X does not occur in A. This leads to introduce the following:

▶ **Definition 26.** *For any polynomial tree* E, *the* characteristic of E, $\kappa(\mathsf{E}) \in \{0, 1, \infty\}$ *is defined as follows: if* E *is coherent, then* $\kappa(\mathsf{E}) = 0$ *if it has no cyclic alternating path, and* $\kappa(\mathsf{E}) = 1$ *if it has a cyclic alternating path; if* E *is not coherent,* $\kappa(\mathsf{E}) = \infty$.

The characteristic is indeed stable under standard reduction (see [26]).

▶ **Lemma 27.** *For all* E, E'*, if* E *reduces to* E' *by standard reduction, then* $\kappa(\mathsf{E}') \leqslant \kappa(\mathsf{E})$.

Using the observation above and Lemma 27 we can easily prove:

▶ **Proposition 28.** *Suppose* $\kappa(\mathsf{E}) \in \{0, 1\}$ *and* E *reduces to* $\mathsf{E}' \in \mathcal{P}_0$ *by standard reduction. If* $\kappa(\mathsf{E}) = 0$*, then* $\tau(\mathsf{E}') \in \Lambda\mathsf{p}$*, and if* $\kappa(\mathsf{E}) = 1$*, then* $\tau(\mathsf{E}') \in \Lambda\mathsf{p}_{\mu\nu}$.

For a $\Lambda 2$-type A, we can define its characteristic as $\kappa(A) = \kappa(\mathsf{t}(A))$. From Prop. 28 and Lemma 16 we deduce then a new criterion for finiteness:

▶ **Corollary 29.** *Let* A *be a closed* $\Lambda 2$*-type. If* $\kappa(A) = 0$*, then* $A \equiv_{\mathsf{Y}} 0 + 1 + \cdots + 1$.

The criterion based on the characteristic can be used to capture yet more finite $\Lambda 2$-types. In fact, whenever a type A reduces to a type with characteristic 0, we can deduce that $A \equiv_{\mathsf{Y}} 0 + 1 + \cdots + 1$. Note that such a type A need not even be coherent. For instance, the (tree of) the type $A = \forall XY.(\forall Z.((Z \Rightarrow Z) \Rightarrow X) \Rightarrow X) \Rightarrow Y \Rightarrow Y$, is not coherent (since the variable Z is not eliminable), but reduces, by eliminating X, to (the tree of) $\forall Y.Y \Rightarrow Y$, which has characteristic 0, and in fact we have that $A \equiv_{\mathsf{Y}} 1$.

As this example shows, a type A can reduce to a finite sum $0 + 1 + \cdots + 1$ even if some of its subtypes cannot be similarly reduced. In fact, while we can eliminate all quantifiers from the type A above, we cannot do this from its subtype $\forall Z.((Z \Rightarrow Z) \Rightarrow X) \Rightarrow X)$. By contrast, in the next section we show that the characteristic satisfies nice compositionality conditions that will allow us to define suitable fragments of $\Lambda 2$.

7 System F with Finite Characteristic

In this section we explore the use of Yoneda reduction to compute program equivalence in $\Lambda 2$. We introduce two fragments of $\Lambda 2$ in which types have a fixed finite characteristic, and we show that the ε-theory for such fragment can be computed by embedding polymorphic programs into well-known monomorphic systems.

First, we have to check that the types with a fixed finite characteristic do yield well-defined fragments of $\Lambda 2$. This requires to check two properties. First, the characteristic has to be *compositional*: a subtype of a type of characteristic k cannot have a higher characteristic, since every subtype of a type of the fragment must be in the fragment itself. Second, since a universally quantified variable can be instantiated with any other type of the fragment, the characteristic must be *closed by instantiation*: if $\forall X.A$ and B have characteristic k, then $A[B/X]$ must have characteristic (at most) k.

▶ **Lemma 30.** ***(compositionality)*** *If* A *is a sub-type of* B*, then* $\kappa(A) \leqslant \kappa(B)$.
(closure by instantiation) $\kappa(A[B/X]) \leqslant \max\{\kappa(\forall X.A), \kappa(B)\}$.

Thanks to Lemma 30 the following fragments can be seen to be well-defined.

▶ **Definition 31** (Systems $\Lambda 2^{\kappa \leqslant k}$)**.** *For* $k = 0, 1$*, let* $\Lambda 2^{\kappa \leqslant k}$ *be the subsystem of* $\Lambda 2$ *with same typing rules and types restricted to the types of* $\Lambda 2$ *of characteristic* k.

We recall that the *free bicartesian closed category* is the category $\mathbb{B} = \mathbb{C}_{\beta\eta}(\Lambda\mathsf{p})$ and the *free cartesian closed* μ*-bicomplete category* $\mu\mathbb{B}$ [29, 6] is the category $\mathbb{C}_{\beta\eta}(\Lambda\mathsf{p}_{\mu\nu})$. The β and η-rules for $\Lambda\mathsf{p}$ and $\Lambda\mathsf{p}_{\mu\nu}$ are recalled in App. A.

$\mathbb{B}$ and $\mu\mathbb{B}$ can be embedded in $\Lambda 2$ by the usual second order encoding (that we note $^\sharp$ and recall in [26]). In fact, it is easily seen that any type of $\Lambda\mathsf{p}$ (resp. of $\Lambda\mathsf{p}_{\mu\nu}$) is encoded by a type of $\Lambda 2^{\kappa \leqslant 0}$ (resp. of $\Lambda 2^{\kappa \leqslant 1}$). Moreover, it is well-known (see [28, 16]) that the η-rules of

$\Lambda\mathsf{p}_{\mu\nu}$ are preserved in $\Lambda 2$ only up to dinaturality (i.e. up to the ε-theory). This embedding yields then a functor $^\sharp : \mu\mathbb{B} \to \mathbb{C}_\varepsilon(\Lambda 2^{\kappa\leqslant 1})$, which restricts to a functor from $\mathbb{B}$ to $\mathbb{C}_\varepsilon(\Lambda 2^{\kappa\leqslant 0})$.

Conversely, from Proposition 28, we already know that the types of $\Lambda 2^{\kappa\leqslant 0}$ (resp. $\Lambda 2^{\kappa\leqslant 1}$) are isomorphic, modulo the ε-theory, to types of $\Lambda\mathsf{p}$ (resp. $\Lambda\mathsf{p}_{\mu\nu}$).

▶ **Proposition 32.** *For all $\Lambda 2^{\kappa\leqslant 0}$-type (resp. $\Lambda 2^{\kappa\leqslant 1}$-type) A there exists a type $A^\flat$ of $\Lambda\mathsf{p}$ (resp. $\Lambda\mathsf{p}_{\mu\nu}$) such that $A \equiv_\varepsilon A^\flat$.*

Proof. Since all isomorphisms $\equiv_Y$ are valid under the ε-theory, we can obtain $A^\flat$ from any standard reduction of the tree of A, using Prop. 28. For technical reasons we will consider a particular reduction, described in detail in [26]. ◀

In [26] we show that the embedding of types above scales to an embedding of terms: for all term t such that $\Gamma \vdash_{\Lambda 2^{\kappa\leqslant 1}} t : A$, we define a term $t^\flat$ such that $\Gamma^\flat \vdash_{\Lambda\mathsf{p}_{\mu\nu}} t^\flat : A^\flat$ holds (with the construction scaling well to $\Lambda 2^{\kappa\leqslant 0}$ and $\Lambda\mathsf{p}$). This yields then a functor $^\flat : \mathbb{C}_\varepsilon(\Lambda 2^{\kappa\leqslant 1}) \to \mu\mathbb{B}$. restricting to a functor from $\mathbb{C}_\varepsilon(\Lambda 2^{\kappa\leqslant 0})$ to $\mathbb{B}$.

The two functors $^\sharp$ and $^\flat$ preserve all the relevant structure (products, coproducts, exponentials, initial algebras, final coalgebras), but they are not *strictly* inverse: $(A^\flat)^\sharp$ is not equal to A, but only ε-isomorphic to it (e.g. for $A = \forall X.((\forall Y.Y \Rightarrow Y) \Rightarrow X) \Rightarrow X$, we have $A^\flat = 1$ and $(A^\flat)^\sharp = \forall X.X \Rightarrow X$). Nevertheless, the following equivalences of categories hold:

▶ **Theorem 33.** $\mathbb{C}_\varepsilon(\Lambda 2^{\kappa\leqslant 0}) \cong \mathbb{B}$, $\mathbb{C}_\varepsilon(\Lambda 2^{\kappa\leqslant 1}) \cong \mu\mathbb{B}$.

The proof of Theorem 33 (in [26]) is done by checking (by way of β-, η- and ε-*rules*, see App. B) that both $\Lambda 2^{\kappa\leqslant 1}$ and $\Lambda\mathsf{p}_{\mu\nu}$ embed *fully* in a suitable fragment of $\Lambda 2\mathsf{p}_{\mu\nu}$, using the lemma below (with $\mathbb{C} = \mathbb{C}_\varepsilon(\Lambda 2^{\kappa\leqslant 1})$, $\mathbb{D} = \mu\mathbb{B}$ and $f(A) = A^\flat$):

▶ **Lemma 34.** *Let $\mathbb{C}, \mathbb{D}$ be full subcategories of a category $\mathbb{E}$. Let $f : \mathsf{Ob}(\mathbb{C}) \to \mathsf{Ob}(\mathbb{D})$ be surjective and suppose there is a map u associating each object a of $\mathbb{C}$ with an isomorphism $u_a : a \to f(a)$ in $\mathbb{E}$. Then f extends to an equivalence of categories $F : \mathbb{C} \to \mathbb{D}$.*

Theorem 33 can be used to deduce properties of program equivalence in $\Lambda 2^{\kappa\leqslant 0}$ and $\Lambda 2^{\kappa\leqslant 1}$ from well-known properties of program equivalence for $\mathbb{B}$ and $\mu\mathbb{B}$. In fact, it is known that $\beta\eta$-equivalence in $\Lambda\mathsf{p}$ (i.e. arrow equivalence in $\mathbb{B}$) is decidable and coincides with contextual equivalence [31], while contextual equivalence for $\mu\mathbb{B}$ is undecidable [6]. Using Theorem 33 we can deduce similar facts for $\Lambda 2^{\kappa\leqslant 0}$ and $\Lambda 2^{\kappa\leqslant 1}$:

▶ **Theorem 35.** *The ε-theory for $\Lambda 2^{\kappa\leqslant 0}$ is decidable and coincides with contextual equivalence. Both the ε-theory and contextual equivalence for $\Lambda 2^{\kappa\leqslant 1}$ are undecidable.*

The first claim of Theorem 35 is proved by defining a new embedding $t \mapsto t^\natural$ of $\Lambda 2^{\kappa\leqslant 0}$ into $\Lambda\mathsf{p}$, exploiting the well-known fact that for the terms of the system $\Lambda 2\mathsf{p}$ one can obtain a normal form under β-reduction and *commutative conversions* [33][3]. Using the isomorphisms $\mathtt{d}_A[x], \mathtt{d}_A^{-1}[x]$ between A and $A^\flat$, a term t such that $\Gamma \vdash_{\Lambda 2^{\kappa\leqslant 0}} t : A$ holds is first translated into $u = \mathtt{d}_A[t[x_i \mapsto \mathtt{d}_{A_i}^{-1}[x_i]]]$ (where $\Gamma = x_1 : A_1, \ldots, x_n : A_n$), and then $t^\natural$ is defined as the normal form of u. From the fact that $\Gamma^\flat \vdash_{\Lambda 2\mathsf{p}} t^\natural : A^\flat$ and that $t^\natural$ is in normal form, we can deduce that $\Gamma^\flat \vdash_{\Lambda\mathsf{p}} t^\natural : A^\flat$ holds, so the embedding is well-defined.

Using the embedding $t \mapsto t^\natural$ we establish the proposition below, from which the first claim of Theorem 35 descends (since $\simeq_{\beta\eta}$ and $\simeq_{\mathsf{ctx}}$ coincide and are decidable in $\Lambda\mathsf{p}$).

[3] Actually, [33] does not consider commuting conversions for 0, but these can be added without altering the existence of normal forms.

▶ **Proposition 36.** $\Gamma \vdash_{\Lambda 2^{\kappa \leqslant 0}} t \simeq_{\varepsilon} u : A$ *iff* $\Gamma \vdash_{\Lambda 2^{\kappa \leqslant 0}} t \simeq_{\mathsf{ctx}} u : A$ *iff* $\Gamma^{\flat} \vdash_{\Lambda \mathsf{p}} t^{\natural} \simeq_{\beta\eta} u^{\natural} : A^{\flat}$.

For the second claim of Theorem 35, the undecidability of contextual equivalence in $\Lambda 2^{\kappa \leqslant 1}$ immediately follows from its undecidability in $\mu\mathbb{B}$ by the encoding $^{\sharp}$. Instead, we do not know if the ε-theory and contextual equivalence coincide in this case, as it is not clear whether the embedding $t \mapsto t^{\natural}$ scales to $\Lambda 2^{\kappa \leqslant 1}$: this depends on the existence of normal forms for commutative conversions, which are not known to hold in presence of μ, ν-types (although this is conjectured in [20]).

The undecidability of the ε-theory is proved in [26] following a different strategy: we first observe that the ε-rules for $\Lambda 2^{\kappa \leqslant 1}$ imply some *uniqueness* rule for inductively/co-inductively defined functions. For instance, under the ε-theory, the usual type of natural numbers $\mathtt{int} = \forall X.(X \Rightarrow X) \Rightarrow (X \Rightarrow X)$ is associated with a uniqueness rule for functions defined by iteration of the following form: for any type C and functions $f : \mathtt{int} \Rightarrow C$ and $h : C \Rightarrow C$, if $f(x+1) \simeq h(f(x))$, then f is equivalent to the function defined by iterating h on $f(0)$, i.e. $f \simeq \lambda x.xCh(f(0))$. Similarly, the type of the usual recursor of $\Lambda 2$ is associated with a uniqueness rule for functions defined by recursion.

It is well-known that any equational theory over a system containing the simply typed λ-calculus, a type of natural numbers and a recursion operator, and satisfying a uniqueness rule for recursively defined functions as above, is undecidable [22]. It suffices then to check that $\Lambda 2^{\kappa \leqslant 1}$, under the ε-theory, provides such a system.

8 Conclusion

Related Work. The connection between parametricity, dinaturality and the Yoneda isomorphism is well-known [5, 28, 16]. The extension of this correspondence to initial algebras comes from [35]. [7] exploits this connection to define a schema to *test* the equivalence of two programs t, u of type $\forall X.(F\langle X\rangle \Rightarrow X) \Rightarrow (G\langle X\rangle \Rightarrow X') \Rightarrow H\langle X\rangle$ by first instantiating X as $\alpha = \mu X.F\langle X\rangle$ and then applying t, u to the canonical morphism $F\langle X \mapsto \alpha\rangle \Rightarrow \alpha$ (in fact, this is exactly how one side of the isomorphisms $\equiv_X$ are constructed). The possibility of expressing program equivalence through naturality conditions has recently attracted new attention due to [3], where these are investigated using ideas from homotopy type theory. Type isomorphisms in $\Lambda 2$ with the Yoneda lemma are also discussed in [17]. In [25] a similar restriction based on the Yoneda isomorphism is used by the first author to describe a fragment of *second order multiplicative linear logic* with a decidable program equivalence.

Future Work. The definition of the characteristic employs an acyclicity condition which is reminiscent of linear logic *proof-nets*. In particular, we would like to investigate whether the alternating paths can be related to the *cyclic proofs* for linear logic systems with μ, ν-types [4]. Moreover, the notion of characteristic seems likely to scale to second order *multiplicative-exponential* linear logic, an extension which might lead to better expose the intrinsic duality in the tree-shapes in Fig. 5.

The application of Yoneda isomorphisms to count type inhabitants suggests that these can be related to some canonical proof-search strategy, as already suggested in [30], that we would like to investigate further. Moreover, the appeal to least/greatest fixpoints suggests a connection with the proof-technique to count inhabitants by computing *fixpoints of polynomial equations* [38]. For example, given $A = (Y_1 \Rightarrow X) \Rightarrow (X \Rightarrow Y_2 \Rightarrow X) \Rightarrow X$, one can show by proof-theoretic reasoning that the number $|A|$ of inhabitants of A is a solution of the fixpoint equation $|A| = |A_{Y_1}| + (|A| \times |A_{Y_2}|)$, where $A_{Y_i} = (Y_1 \Rightarrow X) \Rightarrow (X \Rightarrow Y_2 \Rightarrow X) \Rightarrow Y_i$, which implies $|A| = 0$, since $|A_{Y_i}| = 0$. On the other hand, Yoneda type isomorphisms yield the strikingly similar computation $\forall \vec{Y} X.A \equiv_X \forall \vec{Y}.\mu X.Y_1 + (X \times Y_2) \equiv_{\vec{Y}} \mu X.0 + (X \times 0) \equiv 0$.

References

1 Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. Theorems for free for free: parametricity, with and without types. In *Proceedings of the ACM on Programming Languages*, volume 1 of *ICFP*, page Article No. 39, New York, 2017.

2 T. Aoto. Uniqueness of normal proofs in implicational intuitionistic logic. *Journal of Logic, Language and Information*, 8:217–242, 1999.

3 Steve Awodey, Jonas Frey, and Sam Speight. Impredicative encodings of (higher) inductive types. In *LICS 2018*, 2018.

4 David Baelde, Amina Doumane, and Alexis Saurin. Infinitary proof theory: the multiplicative-additive case. In *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 42:1–42:17, Germany, 2016. Dagstuhl.

5 E.S. Bainbridge, Peter J. Freyd, Andre Scedrov, and Philip J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70:35–64, 1990.

6 Henning Basold and Helle Hvid Hansen. Well-definedness and observational equivalence for inductive-coinductive programs. *Journal of Logic and Computation*, exw091, 2016.

7 Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. Testing Polymorphic Properties. In *ESOP 2010: Programming Languages and Systems*, volume 6012 of *Lecture Notes in Computer Science*, pages 125–144. Springer Berlin Heidelberg, 2010.

8 P. Bourreau and S. Salvati. Game semantics and uniqueness of type inhabitance in the simply-typed λ-calculus. In *TLCA 2011*, volume 6690 of *LNCS*, pages 61–75. Springer Berlin Heidelberg, 2011.

9 S. Broda and L. Damas. On long normal inhabitants of a type. *Journal of Logic and Computation*, 15:353–390, 2005.

10 Kim B. Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 1:1–20, 1991.

11 Joachim de Lataillade. Dinatural terms in System F. In *Proceedings of the Twenty-Fourth Annual IEEE Symposium on Logic in Computer Science (LICS 2009)*, pages 267–276, Los Angeles, California, USA, 2009. IEEE Computer Society Press.

12 Roberto Di Cosmo. A short survey of isomorphisms of types. *Mathematical Structures in Computer Science*, 15:825–838, 2005.

13 Fiore, Roberto Di Cosmo, and Vincent Balat. Remarks on isomorphisms in typed lambda calculi with empty and sum types. *Annals of Pure and Applied Logic*, 141(1–2):35–50, 2006.

14 Nicola Gambino and Joachim Kock. Polynomial functors and polynomial monads. *Mathematical Structures in Computer Science*, 154(1):153–192, 2013.

15 Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Normal forms and cut-free proofs as natural transformations. In Y. Moschovakis, editor, *Logic from Computer Science*, volume 21, pages 217–241. Springer-Verlag, 1992.

16 Ryu Hasegawa. Categorical data types in parametric polymorphism. *Mathematical Structures in Computer Science*, 4(1):71–109, 1994.

17 Ralf Hinze and Daniel W.H. James. Reason isomorphically! In *WGP 2010*, pages 85–96, 2010.

18 Danko Ilik. Axioms and decidability for type isomorphism in the presence of sums. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2014.

19 Patricia Johann. On proving the correctness of program transformation based on free theorems for higher-order polymorphic calculi. *Mathematical Structures in Computer Science*, 15:201–229, 2005.

20 Ralph Matthes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*. PhD thesis, Ludwig-Maximilians-Universität München, 1998.

21 Guy McCuscker and Alessio Santamaria. On compositionality of dinatural transformations. In *CSL 2018*, volume 119 of *LIPIcs*, pages 33:1–33:22, Dagstuhl, Germany, 2018. Schloss Daghstuhl–Leibinz-Zentrum fuer Informatik.

22 M. Okada and P.J. Scott. A note on rewriting theory for uniqueness of iteration. *Theory and Applications of Categories*, 6:47–64, 1999.

23 Paolo Pistone. On dinaturality, typability and $\beta\eta$-stable models. In Schloos Dagstul-Leibniz-Zentrum fuer Informatik, editor, *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)*, volume 84 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:17, Dagstuhl, Germany, 2017.

24 Paolo Pistone. On completeness and parametricity in the realizability semantics of System F. *Logical Methods in Computer Science*, 15(4):6:1–6:54, 2019.

25 Paolo Pistone. Proof nets, coends and the Yoneda isomorphism. In Thomas Ehrhard, Maribel Fernández, Valeria de Paiva, and Lorenzo Tortora de Falco, editors, *Proceedings Joint International Workshop on* Linearity *&* Trends in Linear Logic and Applications *(Linearity-TLLA 2018)*, volume 292 of *EPTCS*, pages 148–167, 2019.

26 Paolo Pistone and Luca Tranchini. The Yoneda Reduction of Polymorphic Types, extended version, 2020. URL: `https://arxiv.org/abs/1907.03481`.

27 Paolo Pistone, Luca Tranchini, and Mattia Petrolo. The naturality of natural deduction (II). Some remarks on atomic polymorphism, 2020. URL: `https://arxiv.org/abs/1908.11353`.

28 Gordon Plotkin and Martin Abadi. A logic for parametric polymorphism. In *TLCA '93, International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375. Springer Berlin Heidelberg, 1993.

29 Luigi Santocanale. Free μ-lattices. *Journal of Pure and Applied Algebra*, 9:166–197, 2002.

30 Gabriel Scherer. *Which types have a unique inhabitant? Focusing on pure program equivalence*. PhD thesis, Université Paris-Diderot, 2016.

31 Gabriel Scherer. Deciding equivalence with sums and the empty type. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 374–386, New York, NY, USA, 2017. ACM.

32 Gabriel Scherer and Didier Rémy. Which simple types have a unique inhabitant? *SIGPLAN Not.*, 50(9):243–255, 2015.

33 Makoto Tatsuta. Second order permutative conversions with Prawitz's strong validity. *Progress in Informatics*, 2:41–56, 2005.

34 Luca Tranchini, Paolo Pistone, and Mattia Petrolo. The naturality of natural deduction. *Studia Logica*, 107(1):195–231, 2019.

35 Tarmo Uustalu and Varmo Vene. The Recursion Scheme from the Cofree Recursive Comonad. *Electronic Notes in Theoretical Computer Science*, 229(5):135–157, 2011.

36 Janis Voigtländer. Free theorems simply, via dinaturality. In *Declarative Programming and Knowledge Management*, pages 247–267, Cham, 2020. Springer International Publishing.

37 Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on functional programming languages and computer architecture - FPCA '89*, 1989.

38 Marek Zaoinc. Fixpoint technique for counting terms in typed lambda-calculus. Technical report, State University of New York, 1995.

A Type Systems, Type Isomorphisms and Equational Theories

The typing rules and β-, η-rules of $\Lambda 2, \Lambda 2\mathsf{p}, \Lambda 2\mathsf{p}_{\mu\nu}$ are recalled in Fig. 8. The standard axiomatization of $\beta\eta$-isomorphisms for $\Lambda 2$ and $\Lambda 2\mathsf{p}$ are recalled in Fig. 9 and 10, and a "minimalistic" axiomatization of $\beta\eta$-isomorphisms for μ, ν-types is illustrated in Fig. 11.

The *contextual equivalence* relation for $\Lambda 2\mathsf{p}_{\mu\nu}$ and $\Lambda\mathsf{p}$ is defined by

$$\Gamma \vdash t \simeq_{\mathsf{ctx}} u : A \text{ iff for all context } \mathtt{C} : (\Gamma \vdash A) \Rightarrow (\vdash 1 + 1),\ \mathtt{C}[t] \simeq_{\beta\eta} \mathtt{C}[u]$$

The contextual equivalence relation for $\Lambda 2$ is defined by

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \Lambda X.t : \forall X.A}\ X \notin FV(\Gamma) \qquad \frac{\Gamma \vdash t : \forall X.A}{\Gamma \vdash tB : A[B/X]}$$

(a) Typing rules for $\Lambda 2$.

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u\rangle : A \times B} \qquad \frac{\Gamma \vdash t : A_1 \times A_2}{\Gamma \vdash \pi_i^{A_i} t : A_i} \qquad \frac{}{\Gamma \vdash \star : 1}$$

$$\frac{\Gamma \vdash t : A_i}{\Gamma \vdash \iota_i t : A_1 + A_2} \qquad \frac{\Gamma \vdash t : A_1 + A_2 \qquad (\Gamma, y : A_i \vdash u_i : C)_{i=1,2}}{\Gamma \vdash \delta_C(t, y.u_1, y.u_2) : C} \qquad \frac{\Gamma \vdash t : 0}{\Gamma \vdash \xi_A t : A}$$

$$\frac{\Gamma \vdash t : A\langle X \mapsto \mu X.A\langle X\rangle\rangle}{\Gamma \vdash \mathsf{in}_A t : \mu X.A\langle X\rangle} \qquad \frac{\Gamma \vdash t : A\langle X \mapsto B\rangle \Rightarrow B}{\Gamma \vdash \mathsf{fold}_A(t) : \mu X.A\langle X\rangle \Rightarrow B}$$

$$\frac{\Gamma \vdash t : \nu X.A\langle X\rangle}{\Gamma \vdash \mathsf{out}_A t : A\langle X \mapsto \nu X.A\langle X\rangle\rangle} \qquad \frac{\Gamma \vdash t : B \Rightarrow A\langle X \mapsto B\rangle}{\Gamma \vdash \mathsf{unfold}_A(t) : B \Rightarrow \nu X.A\langle X\rangle}$$

(b) Typing rules for $+, \times, 0, 1, \mu, \nu$.

$$(\lambda x.t)u \ \simeq_\beta \ t[u/x] \qquad (\Lambda X.t)B \ \simeq_\beta \ t[B/X]$$

$$\frac{\Gamma \vdash t : A \to B}{\Gamma \vdash t \simeq_\eta \lambda x.tx : A \to B} \qquad \frac{\Gamma \vdash t : \forall X.A}{\Gamma \vdash t \ \simeq_\eta \ \Lambda X.tX : \forall X.A}$$

(c) β and η-rules for $\Lambda 2$.

$$\left(\pi_i^{A_i}\langle t_1, t_2\rangle \ \simeq_\beta \ t_i\right)_{i=1,2} \qquad \left(\delta_C(\iota_i t, y.u_1, y.u_2) \ \simeq_\beta \ u_i[t/y]\right)_{i=1,2}$$

$$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash t \simeq_\eta \langle \pi_1^A t, \pi_2^B t\rangle : A \times B} \qquad \frac{\Gamma \vdash t : 1}{\Gamma \vdash t \simeq_\eta \star : 1}$$

$$\frac{\Gamma \vdash t : A + B \qquad u[x] : A + B \vdash^\Gamma C}{\Gamma \vdash u[t] \simeq_\eta \delta_C(t, y.u[\iota_1 y], y.u[\iota_2 y]) : C} \qquad \frac{\Gamma \vdash t : 0 \qquad u[x] : 0 \vdash^\Gamma A}{\Gamma \vdash u[t] \simeq_\eta \xi_A t : A}$$

(d) β and η-rules for $+, \times, 0, 1$.

$$\mathsf{fold}_P(t)(\mathsf{in}_P u) \simeq_\beta t(\Phi_P^X(\mathsf{fold}_P(t)x)[x \mapsto u])$$

$$\mathsf{out}_P(\mathsf{unfold}_P(t)u) \simeq_\beta \Phi_P^X(\mathsf{unfold}_P(t)x)[x \mapsto tu]$$

$$\frac{\Gamma \vdash u : A\langle X \mapsto C\rangle \Rightarrow C \qquad t[x] : \mu X.A\langle X\rangle \vdash^\Gamma C \qquad t[\mathsf{in}_A x] \simeq u\Phi_A^X(t) : A\langle X \mapsto \mu X.A\langle X\rangle\rangle \vdash^\Gamma C}{t[x] \simeq_\eta \mathsf{fold}_A(u)x : \mu X.A\langle X\rangle \vdash^\Gamma C}$$

$$\frac{\Gamma \vdash u : C \Rightarrow A\langle X \mapsto C\rangle \qquad t[x] : C \vdash^\Gamma \nu X.A\langle X\rangle \qquad \Phi_A^X(t)[x \mapsto ux] \simeq \mathsf{out}_A t : C \vdash^\Gamma A\langle X \mapsto \nu XA\langle X\rangle\rangle}{t[x] \simeq_\eta \mathsf{unfold}_A(u)x : C \vdash^\Gamma \nu X.A\langle X\rangle}$$

(e) β and η-rules for μ, ν.

Figure 8 Typing rules and $\beta\eta$-rules.

$$A \Rightarrow (B \Rightarrow C) \equiv B \Rightarrow (A \Rightarrow C)$$

$$\forall X.\forall Y.A \equiv \forall Y.\forall X.A$$

$$A \equiv \forall X.A \quad (X \notin FV(A)) \qquad A \Rightarrow \forall X.B \equiv \forall X.A \Rightarrow B$$

Figure 9 Axiomatization of $\beta\eta$-isomorphisms for $\mathsf{\Lambda 2}$.

$$A \times 1 \equiv A \qquad 1 \Rightarrow A \equiv A$$

$$A \times (B \times C) \equiv (A \times B) \times C \qquad A \times B \equiv B \times A$$

$$(A \times B) \Rightarrow C \equiv A \Rightarrow (B \Rightarrow C) \qquad \forall X.A \times B \equiv \forall X.A \times \forall X.B$$

$$A + 0 \equiv A \quad A \times 0 \equiv 0 \qquad 0 \Rightarrow A \equiv 1$$

$$A + (B + C) \equiv (A + B) + C \qquad A + B \equiv B + A$$

$$A \times (B + C) \equiv (A \times B) + (A \times C) \qquad (A + B) \Rightarrow C \equiv (A \Rightarrow C) \times (B \Rightarrow C)$$

Figure 10 Axiomatization of $\beta\eta$-isomorphisms for $\mathsf{\Lambda 2p}$.

$$\Gamma \vdash t \simeq_{\mathsf{ctx}} u : A \text{ iff for all context } \mathtt{C} : (\Gamma \vdash A) \Rightarrow (\vdash \forall X.X \Rightarrow X \Rightarrow X),\ \mathtt{C}[t] \simeq_{\beta\eta} \mathtt{C}[u]$$

It is a standard result that contextual equivalence (for either $\mathsf{\Lambda 2p}_{\mu\nu}$ or $\mathsf{\Lambda 2}$) is closed under congruence rules and thus generates an equational theory ctx.

For all positive type $A\langle X\rangle$ and negative type $B\langle X\rangle$ the functors $\Phi^X_A : \mathbb{C}_{\beta\eta}(\mathsf{\Lambda 2}) \to \mathbb{C}_{\beta\eta}(\mathsf{\Lambda 2})$ and $\Phi^X_B : \mathbb{C}_{\beta\eta}(\mathsf{\Lambda 2})^{\mathsf{op}} \to \mathbb{C}_{\beta\eta}(\mathsf{\Lambda 2})$ are defined by letting $\Phi^X_A(C) = A[C/X]$, $\Phi^X_B(C) = B[C/X]$ and for all $t[x] : C \vdash D$,

$$\Phi^X_X(t) = t \tag{5}$$

$$\Phi^X_Y(t) = x \tag{6}$$

$$\Phi^X_{(C \Rightarrow D)}(t) = \lambda y.\Phi^X_D(t[\Phi^X_C(y)]) \tag{7}$$

$$\Phi^X_{(\forall X.C)}(t) = \Lambda Y.\Phi^X_C(tY) \tag{8}$$

One can check that $\Phi^X_A(x) \simeq_\eta x$ and $\Phi^X_A(t[u[x]]) \simeq_\beta \Phi^X_A(t)\Big[x \mapsto \Phi^X_A(u)\Big]$.

The definition above can be extended to the types defined using all other constructors, yielding functors $\Phi^X_A : \mathbb{C}_{\beta\eta}(\mathsf{\Lambda 2p}_{\mu\nu}) \to \mathbb{C}_{\beta\eta}(\mathsf{\Lambda 2p}_{\mu\nu})$ and $\Phi^X_B : \mathbb{C}_{\beta\eta}(\mathsf{\Lambda 2p}_{\mu\nu})^{\mathsf{op}} \to \mathbb{C}_{\beta\eta}(\mathsf{\Lambda 2p}_{\mu\nu})$.

B The ε-Theory and the Yoneda Isomorphisms

In this section we describe the equational theory induced by the interpretation of polymorphic programs as dinatural transformations (for a suitable fragment of $\mathsf{\Lambda 2p}_{\mu\nu}$), that we call the ε-theory, and we show that the type isomorphisms $\equiv_X$ and $\equiv_X$ from Section 2 hold under this theory.

We will work for simplicity in an *ad-hoc* fragment $\mathsf{\Lambda 2p}^*_{\mu,\nu}$ of $\mathsf{\Lambda 2p}_{\mu\nu}$, in which we require

$$\mu X.A \equiv \nu X.A \equiv A \quad (X \notin FV(A))$$

$$\mu X.A\langle X\rangle \equiv A\langle X \mapsto \mu X.A\langle X\rangle\rangle \qquad \nu X.A\langle X\rangle \equiv A\langle X \mapsto \nu X.A\langle X\rangle\rangle$$

Figure 11 Axiomatization of $\beta\eta$-isomorphisms for μ,ν-types.

that all universal types are of one of the two forms below:

$$\forall X.\Big\langle \forall \vec{Y}_k.\big\langle A_{jk}\langle X\rangle\big\rangle_j \Rightarrow X\Big\rangle_k \Rightarrow B\langle X\rangle \qquad \forall X.\Big\langle \forall \vec{Y}_j.X \Rightarrow A_j\langle X\rangle\Big\rangle_j \Rightarrow B\langle X\rangle \quad (9)$$

As this set of types is stable by substitution, all type rules and equational rules of $\Lambda 2\mathsf{p}_{\mu\nu}$ scale well to $\Lambda 2\mathsf{p}^*_{\mu,\nu}$.

To each universal type $\forall X.C$ of $\Lambda 2\mathsf{p}^*_{\mu,\nu}$ as in Eq. (9) left (resp. Eq. (9) right) we associate the *ε-rule* illustrated in Fig. 12a (resp. Fig. 12b).[4] From the viewpoint of category theory, the two ε-rules express *strong dinaturality* conditions [35] for the transformations induced by polymorphic programs (illustrated in Fig. 12c and Fig. 12d).

▶ **Definition 37** (ε-theory). *The* ε-theory *of* $\Lambda 2\mathsf{p}^*_{\mu,\nu}$ *is the smallest congruent equational theory closed under β-, η-equations as well as ε-rules.*

We will show that the isomorphism schema $\equiv_X$, that we recall below, holds under the ε-theory (a similar argument can be developed for the isomorphism schema $\equiv_X$, see [26]).

$$\forall X.\Big\langle \forall \vec{Y}_k.\big\langle A_{jk}\langle X\rangle\big\rangle_j \Rightarrow X\Big\rangle_k \Rightarrow B\langle X\rangle \equiv B\left\langle X \mapsto \{\mu X.\}\sum_k\left(\exists \vec{Y}_k.\prod_j A_{jk}\langle X\rangle\right)\right\rangle \quad (10)$$

▶ **Notation B.1.** *Let $L = \langle i_1,\dots,i_k\rangle$ be a list. If $(t_k)_{i\in L}$ is a L-indexed list of terms, we let for any term u, $u\langle t_k\rangle_{i\in L}$ be shorthand for $ut_{i_1}\dots t_{i_k}$; if $\langle x_{i_1},\dots,x_{i_k}\rangle$ is a L-indexed list of variables, we let for any term u, $\lambda\langle x_k\rangle_{i\in L}.u$ be shorthand for $\lambda x_{i_1}\dots\lambda x_{i_k}.u$.*

In the case of Eq. (10) we can construct terms

$$\mathsf{a}_k[\langle z_j\rangle_j] = \mathtt{in}_k^{\sharp K}(\mathsf{pack}[\vec{Y}_k](\mathtt{prod}_j^{\sharp J_k}\langle z_j\rangle_j)) \;:\; \big\langle A_{jk}\langle X\mapsto\alpha\rangle\big\rangle_j \vdash T\langle X\mapsto\alpha\rangle$$
$$\hat{\mathsf{a}}_k = \Lambda\vec{Y}_k.\lambda\langle z_j\rangle_j.\mathtt{in}_T(\mathsf{a}_k[\langle z_j\rangle_j]) \;:\; \forall\vec{Y}_k.\langle A_{jk}\langle X\mapsto\alpha\rangle\rangle_j \Rightarrow \alpha$$

where $T\langle X\rangle = \sum_k \exists\vec{Y}_k.\prod_j A_{jk}\langle X\rangle$ and $\alpha = \mu X.T\langle X\rangle$, $\mathtt{in}_k^{\sharp I} : X_k \Rightarrow \sum_k^{\sharp I} X_k$ and $\mathtt{prod}_j : \langle X_j\rangle_j \Rightarrow \prod_j^{\sharp J_k} X_j$ are defined composing usual sum and product constructors, and pack is defined as follows:

$$\mathsf{pack}[B_1,\dots,B_k] = \lambda x.\Lambda Z.\lambda f.xB_1\dots B_k f : A[B_1/Y_1,\dots,B_k/Y_k] \Rightarrow \exists\vec{Y}.A$$

With such terms we can then construct a term

$$\mathtt{s}_{A_{jk},B}[x] = x\langle\hat{\mathsf{a}}_k\rangle_k : \Big\langle\forall\vec{Y}_k.\big\langle A_{jk}\langle X\mapsto\alpha\rangle\big\rangle_j\Rightarrow\alpha\Big\rangle_k \Rightarrow B\langle X\mapsto\alpha\rangle \quad\vdash\quad B\langle X\mapsto\alpha\rangle$$

Moreover, using sum and product destructors we can construct terms

$$\mathtt{b}[x,Z] \;:\; T\langle X\mapsto Z\rangle \vdash^{\Delta} Z$$
$$\mathtt{t}_{A_{jk},B}[x,Z] \;:\; B\langle X\mapsto\alpha\rangle \quad\vdash\quad \Big\langle\forall\vec{Y}_k.\big\langle A_{jk}\langle X\mapsto Z\rangle\big\rangle_j\Rightarrow Z\Big\rangle_k \Rightarrow B\langle X\mapsto Z\rangle$$

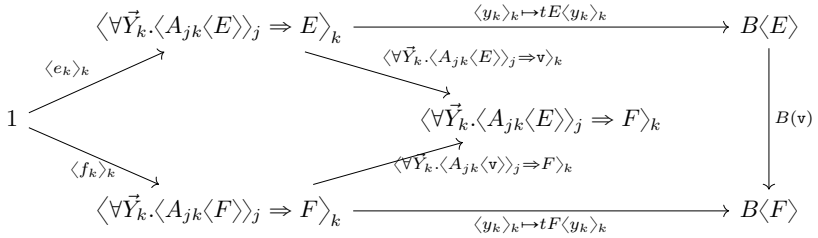
[4] Observe that $\forall X.C$ might well be of *both* forms (9) left and right

$$\frac{\begin{array}{c} t : \forall X.C \\ \left(e_k : \forall \vec{Y}_k.\langle A_{jk}\langle E\rangle\rangle_j \Rightarrow E\right)_k \\ \left(f_k : \forall \vec{Y}_k.\langle A_{jk}\langle F\rangle\rangle_j \Rightarrow F\right)_k \\ \mathrm{v}[x] : E \vdash F \\ \mathrm{v}[e_k\vec{Y}_k\langle z_j\rangle_j] \simeq f_k\vec{Y}_k\langle \Phi^X_{A_{jk}}(\mathrm{v})[z_j]\rangle_j : \langle A_{jk}\langle X \mapsto E\rangle\rangle_j \vdash F \end{array}}{\Gamma \vdash \Phi^X_B(\mathrm{v})\left[x \mapsto tE\langle e_k\rangle_k\right] \simeq tF\langle f_k\rangle_k : B\langle X \mapsto F\rangle}$$

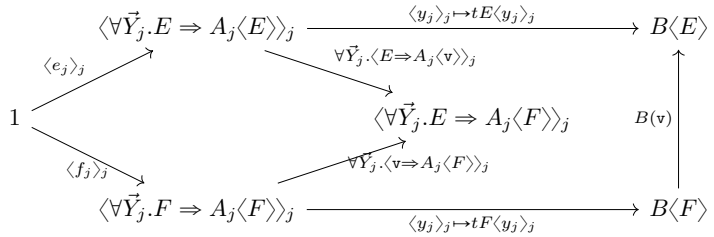
(a) ε-rule for the left-hand type in Eq. (9).

$$\frac{\begin{array}{c} t : \forall X.C \\ \left(e_j : \forall \vec{Y}_k.E \Rightarrow A_j\langle E\rangle\right)_j \\ \left(f_j : \forall \vec{Y}_k.F \Rightarrow A_j\langle F\rangle\right)_j \\ \mathrm{v}[x] : E \vdash F \\ \Phi^X_{A_j}(\mathrm{v}[x])[e_j x] \simeq f_j \mathrm{v}[x] : E \vdash A_j\langle X \mapsto F\rangle \end{array}}{\Gamma \vdash \Phi^X_B(\mathrm{v})\left[x \mapsto tF\langle f_j\rangle_j\right] \simeq tE\langle e_j\rangle_j : B\langle X \mapsto E\rangle}$$

(b) ε-rule for the left-hand type in Eq. (9).



(c) Strong dinaturality diagram for the ε-rule **(a)**.



(d) Strong dinaturality diagram for the ε-rule **(b)**.

Figure 12 ε-rules and their associated strong dinaturality diagrams.

where $\Delta = \{\langle f_k : \forall \vec{Y}_k.\langle A_{jk}\langle X \mapsto Z\rangle\rangle_j \Rightarrow Z\rangle_k\}$ and

$$\begin{aligned} \mathsf{b}[x, Z] &= \delta^{\sharp K}\left(x, \left\langle z.\mathsf{unpack}(z)\left(\Lambda\vec{Y}_k.\lambda y.f_k\vec{Y}_k\langle \pi^{\sharp J_k}_j(y)\rangle_j\right)\right\rangle_k\right) \\ \mathsf{t}_{A_{jk},B}[x, Z] &= \lambda\langle f_k\rangle_k.\Phi^X_B\left(\mathsf{fold}_T(\lambda x.\mathsf{b}[x, Z])x\right) \end{aligned}$$

where π^j_i and $\delta^{\sharp K}(t, \langle z.u_k\rangle_k)$ indicate suitable generalized product and sum destructors which can be defined inductively using product and sum destructors, and the term unpack is defined as follows:

$$\mathsf{unpack} = \Lambda Z.\lambda x.\lambda f.fZx : \forall Z.\left(\exists \vec{Y}.A\right) \Rightarrow (\forall \vec{Y}.A \Rightarrow Z) \Rightarrow Z$$

One can check that $\mathtt{b}[x,\alpha]\big[\langle f_k \mapsto \hat{\mathtt{a}}_k\rangle_k\big] : T\langle X \mapsto \alpha\rangle \vdash \alpha$ is $\beta\eta$-equivalent to $\mathsf{in}_T x$, from which we deduce

$$\mathsf{fold}_T\Big(\lambda x.\mathtt{b}[x,\alpha]\big[\langle f_k \mapsto \hat{\mathtt{a}}_k\rangle_k\big]\Big)x \simeq_{\beta\eta} \mathsf{fold}_T(\lambda x.\mathsf{in}_T x)x \simeq_\eta x$$

In this way the isomorphism $\equiv_X$ are realized in $\mathbb{C}_\varepsilon(\Lambda 2\mathsf{p}^*_{\mu,\nu})$ by the two terms below:[5]

$$\mathtt{s}[x] = \mathtt{s}_{A_{jk},B}[x\alpha] \qquad\qquad \mathtt{t}[x] = \Lambda X.\mathtt{t}_{A_{jk},B}[x,X]$$

We can compute then

$$\mathtt{s}[\mathtt{t}[x]] \simeq_\beta \Phi^X_B\Big(\mathsf{fold}_T\big(\lambda x.\mathtt{b}[x,\alpha]\big[\langle f_k \mapsto \langle \hat{\mathtt{a}}_k\rangle_k\big]\big)x\Big) \simeq_{\beta\eta} \Phi^X_B(x) \simeq_\eta x$$

and

$$\begin{aligned}\mathtt{t}[\mathtt{s}[x]] &\simeq_\beta \Lambda X.\lambda\langle f_k\rangle_k.\Phi^X_B\Big(\mathsf{fold}_T(\lambda x.\mathtt{b}[x,X])x\Big)\Big[x \mapsto x\alpha\langle \hat{\mathtt{a}}_k\rangle_k\Big]\\ &\simeq_\varepsilon \Lambda X.\lambda\langle f_k\rangle_k.xX\langle f_k\rangle_k \simeq_\eta x\end{aligned}$$

where the central ε-equivalence is justified using the ε-rule in Fig. 12a with $E=\alpha$, $F=X$, $e_k = \hat{\mathtt{a}}_k$ and $\mathtt{v}[x] = \mathsf{fold}_T(\lambda x.\mathtt{b}[x,X])x$, with the last premise given by the computation below:

$$\Big(\mathsf{fold}_T(\lambda x.\mathtt{b}[x,X])x\Big)\Big[x \mapsto \hat{\mathtt{a}}_k\vec{Y}_k\langle z_j\rangle_j\Big] \simeq_\beta \Big(\mathsf{fold}_T(\lambda x.\mathtt{b}[x,X])\Big)\mathsf{in}_T(\mathtt{a}_k[\langle z_j\rangle_j])$$

$$\simeq_\beta \big(\lambda x.\mathtt{b}[x,X]\big)\Big(\big(\Phi^X_T(\mathsf{fold}_T(\lambda x.\mathtt{b}[x,X])x)\big)\Big[x \mapsto \mathtt{a}_k[\langle z_j\rangle_j]\Big]\Big)$$

$$\simeq_\beta f_k\vec{Y}_k\Big\langle \Phi^X_{A_{jk}}(\mathsf{fold}_T(\lambda x.\mathtt{b}[x,X])x)[x \mapsto z_j]\Big\rangle_j$$

where the last β-equation can be checked by unrolling the definition of Φ^X_T:

$$\Phi^X_T(t) = \delta^{\sharp K}\Big(x, \Big\langle z.\mathsf{unpack}(z)\big(\Lambda\vec{Y}_k.\lambda x.\mathtt{in}^{\sharp K}_k(\mathsf{pack}[\vec{Y}_k](\mathtt{prod}^{\sharp J_k}\langle \Phi^X_{A_{jk}}(t[x \mapsto \pi^{\sharp J_k}_j(x)])\rangle_j))\big)\Big\rangle_k\Big)$$

[5] We are here supposing that X does occur in at least some of the A_{jk} (so that $\mu X.$ actually occurs in the left-hand type of $\equiv_X$). If this is not the case, the construction can be done in a similar (and simpler) way.