On the exp-log normal form of types

Danko Ilik

To cite this version:

Danko Ilik. On the exp-log normal form of types. 2015. hal-01167162v1

HAL Id: hal-01167162

https://inria.hal.science/hal-01167162v1

Preprint submitted on 23 Jun 2015 (v1), last revised 17 Aug 2016 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

On the exp-log normal form of types

Danko Ilik

June 23, 2015

Abstract

We present a type pseudo-normal form that any type built from arrows, products and sums can be isomorphically mapped to and that systematically minimizes the number of premises of sum type used. Inspired from a representation of exponential polynomials, the normal form presents an extension of the notion of disjunctive normal form that handles arrows. We also show how to apply it for simplifying the axioms of the theory of $\beta\eta$-equality of terms of the lambda calculus with sums.

1 Introduction

The lambda calculus is a notation for writing functions. Be it simply-typed or polymorphic, it is also often presented as the core of modern functional programming languages. Yet, besides functions as first-class objects, another essential ingredient of these languages are algebraic data types that typing systems supporting only the $\rightarrow$-type and polymorphism do not model so well. A natural model for the core of functional languages should at least include direct support for a simplest case of variant types – sums. However, the theory of the $\{\rightarrow, +\}$-typed lambda calculus, unlike the theory of the $\rightarrow$-typed one, is not all roses.

Take for example the term $\lambda xy.yx$ of type $\tau + \sigma \rightarrow (\tau + \sigma \rightarrow \rho) \rightarrow \rho$. Which of the following three candidates should be its canonical η-*long* β-normal form?

$$\lambda x.\lambda y.y\delta(x, z.\iota_1 z, z.\iota_2 z)$$
$$\lambda x.\lambda y.\delta(x, z.y(\iota_1 z), z.y(\iota_2 z))$$
$$\lambda x.\delta(x, z.\lambda y.y(\iota_1 z), z.\lambda y.y(\iota_2 z))$$

All three are β-normal, η-long, and can be proven mutually equal using the standard equational theory of $=_{\beta\eta}$ (Figure 3), but why should we prefer any one of them over the others? Or consider the following two terms of type $(\tau_1 \rightarrow \tau_2) \rightarrow (\tau_3 \rightarrow \tau_1) \rightarrow \tau_3 \rightarrow \tau_4 + \tau_5 \rightarrow \tau_2$ (taken from [4]):

$$\lambda xyzu.x(yz)$$
$$\lambda xyzu.\delta(\delta(u, x_1.\iota_1 z, x_2.\iota_2(yz)), y_1.x(yy_1), y_2.xy_2).$$

Although a relatively simple example of two $\beta\eta$-equal terms, is it immediate to notice the equality? More generally, are we able to write a program to decide for any given two terms of the lambda calculus with sums whether they are $\beta\eta$-equal?

As far as I can tell, this problem is still open. In spite of significant contributions to the understanding of (non-)canonicity of terms and the equational theory of the calculus [9, 10, 12, 2, 8, 4, 15, 1, 16], we still do not have theoretical understanding that can lead to a (simple) implementation of a decision procedure for $\beta\eta$-equality of terms. The only publicly available implementation of such a procedure, the one of Balat [3], is not complete, for it relies on building normal forms of terms that are canonical only up to a non-trivial equivalence relation.

If we leave aside the problem of equality of terms, there is another problem with sums, at the level of types, where in spite of recent meta-theoretic work [11, 14] on type isomorphisms, we still have to find solutions for isomorphism building that would be implementable in practice.

The first goal of this paper is to introduce a new theoretical device, a pseudo normal form for types build from $\{\rightarrow, +, \times\}$, called the exp-log normal form (Section 2). The second goal is to apply the type normal form to the problem of $\beta\eta$-equality of terms of the corresponding lambda calculus (Section 3). We believe the type normal form to be of more general interest as explained in the conclusion (Section 4).

2 The exp-log normal form

The trouble with sums starts already at the level of types. Namely, when we consider types built from function spaces, products, and disjoint unions (sums),

$$\tau, \sigma ::= \chi_i \mid \tau \rightarrow \sigma \mid \tau \times \sigma \mid \tau + \sigma,$$

where χ_i are base types (or type variables), it is not always clear when two given types are essentially the same one. More precisely, it is not always known how to decide *effectively* whether two types are isomorphic [14]. Although the notion of isomorphism can be treated abstractly in Category Theory and without committing to a specific term calculus inhabiting the types, in the language of the standard syntax and equational theory of lambda calculus with sums (Figure 3), the types τ and σ are isomorphic when there exist coercing lambda terms $M : \sigma \rightarrow \tau$ and $N : \tau \rightarrow \sigma$ such that

$$\lambda x.M(Nx) =_{\beta\eta} \lambda x.x \quad \text{and} \quad \lambda y.N(My) =_{\beta\eta} \lambda y.y.$$

In other words, data/programs can be converted back and forth between τ and σ without loss of information.

The problem of isomorphism is in fact closely related [11] to the famous Tarski High School Identities Problem [6, 7]; for a recent survey on the connection, see [14]. What is important for us here is that *types can be seen as just arithmetic expressions*: if the type $\tau \rightarrow \sigma$ is denoted by the binary arithmetic exponentiation σ^τ, then every type ρ denotes at the same time an *exponential* polynomial ρ. The difference with ordinary polynomials is that the exponent can now also contain a (type) variable, while

exponentiation in ordinary polynomials is always of the form σ^n for a concrete $n \in \mathbb{N}$ i.e. $\sigma^n = \underbrace{\sigma \times \cdots \times \sigma}_{n\text{-times}}$. Moreover, we have that

$$\tau \cong \sigma \text{ implies } \mathbb{N}^+ \vDash \tau = \sigma,$$

that is, type isomorphism implies that arithmetic equality holds for any substitution of variables by positive natural numbers.

This could hence provide an effective *non*-isomorphism decision procedure: given any two types, prove they are not equal as exponential polynomials, and that means they cannot possibly be isomorphic. But, we are interested in a positive decision procedure. Such a procedure exists for both the languages of types $\{\to, \times\}$ and $\{\times, +\}$, since then we have an equivalence:

$$\tau \cong \sigma \text{ iff } \mathbb{N}^+ \vDash \tau = \sigma.$$

Indeed, in these cases type isomorphism can not only be decided, but also effectively built. In the case of $\{\times, +\}$, the procedure amounts to transforming the type to disjunctive normal form, or the (*non*-exponential) polynomial to canonical form, while in that of $\{\to, \times\}$, there is a normal form obtained by type transformation that follows currying [5].

Given that we do not know whether it is possible to find such a normal form for the full language of types [14], what we can hope to do in practice is to find at least a *pseudo* normal form. We shall now define such a type normal form and later, in Section 3, show its beneficial effects for the theory of $=_{\beta\eta}$.

The idea is to use the decomposition of the binary exponential function σ^τ through unary exponentiation and logarithm. This is a well known transformation in Analysis, where for the natural logarithm and Euler's number e we would use

$$\sigma^\tau = \mathrm{e}^{\tau \times \log \sigma} \quad \text{also written} \quad \sigma^\tau = \exp(\tau \times \log \sigma).$$

The systematic study of such normal forms by Du Bois-Reymond described in the book [13] served us as inspiration.

But how exactly are we to go about using this equality for types when it uses logarithms i.e. transcendental numbers? Luckily, we do not have to think of real numbers at all, because what is described above can be seen through the eyes of abstract Algebra, in exponential fields, as a pair of mutually inverse homomorphisms $\exp$ and $\log$ between the multiplicative and additive group, satisfying

$$\begin{aligned} \exp(\tau_1 + \tau_2) &= \exp \tau_1 \times \exp \tau_2 & \qquad \exp(\log \tau) &= \tau \\ \log(\tau_1 \times \tau_2) &= \log \tau_1 + \log \tau_2 & \qquad \log(\exp \tau) &= \tau. \end{aligned}$$

In other words, $\exp$ and $\log$ can be considered as macro expansions rather than unary type constructors. Let us take the type $\tau + \sigma \to (\tau + \sigma \to \rho) \to \rho$ from Section 1, assuming for simplicity that τ, σ, ρ are base types. It can be normalized in the following

way:

$$
\begin{aligned}
&\tau + \sigma \to (\tau + \sigma \to \rho) \to \rho \\
=&\left(\rho^{\rho^{\tau+\sigma}}\right)^{\tau+\sigma} \\
=&\exp((\tau + \sigma)\log[\exp\{\exp((\tau + \sigma)\log \rho)\log \rho\}]) \\
\rightsquigarrow&\exp((\tau + \sigma)\log[\exp\{\exp(\tau \log \rho)\exp(\sigma \log \rho)\log \rho\}]) \\
\rightsquigarrow&\exp((\tau + \sigma)\exp(\tau \log \rho)\exp(\sigma \log \rho)\log \rho) \\
\rightsquigarrow&\exp(\tau \exp(\tau \log \rho)\exp(\sigma \log \rho)\log \rho)\exp(\sigma \exp(\tau \log \rho)\exp(\sigma \log \rho)\log \rho) \\
=&\rho^{\tau\rho^{\tau}\rho^{\sigma}}\rho^{\sigma\rho^{\tau}\rho^{\sigma}} \\
=&(\tau \times (\tau \to \rho) \times (\sigma \to \rho) \to \rho) \times (\sigma \times (\tau \to \rho) \times (\sigma \to \rho) \to \rho).
\end{aligned}
$$

As the exp-log transformation of arrow types is at the source of this type normalization procedure, we call the obtained normal form *the exp-log normal form (ENF)*. However, all this transformation does is that it *orients* the high-school identities,

$$
\begin{array}{ccc}
(f+g)+h \rightsquigarrow f+(g+h) & \text{i.e.} & (f+g)+h \rightsquigarrow f+(g+h) \\
(fg)h \rightsquigarrow f(gh) & \text{i.e.} & (f \times g) \times h \rightsquigarrow f \times (g \times h) \\
f(g+h) \rightsquigarrow fg+fh & \text{i.e.} & f \times (g+h) \rightsquigarrow f \times g + f \times h \\
(f+g)h \rightsquigarrow fh+gh & \text{i.e.} & (f+g) \times h \rightsquigarrow f \times h + g \times h \\
f^{g+h} \rightsquigarrow f^g f^h & \text{i.e.} & (g+h) \to f \rightsquigarrow (g \to f) \times (h \to f) \\
(fg)^h \rightsquigarrow f^h g^h & \text{i.e.} & h \to f \times g \rightsquigarrow (h \to f) \times (h \to g) \\
(f^g)^h \rightsquigarrow f^{hg} & \text{i.e.} & h \to (g \to f) \rightsquigarrow h \times g \to f,
\end{array}
$$

all of which are valid as type isomorphisms. We can thus also compute the *isomorphic* normal form of the type directly, for instance for the second example of Section 1:

$$
\begin{aligned}
&(\tau_1 \to \tau_2) \to (\tau_3 \to \tau_1) \to \tau_3 \to \tau_4 + \tau_5 \to \tau_2 \\
=&\left(\left(\left(\tau_2^{\tau_4+\tau_5}\right)^{\tau_3}\right)^{{\tau_1}^{\tau_3}}\right)^{{\tau_2}^{\tau_1}} \rightsquigarrow \tau_2^{{\tau_2}^{\tau_1}{\tau_1}^{\tau_3}\tau_3\tau_4}\tau_2^{{\tau_2}^{\tau_1}{\tau_1}^{\tau_3}\tau_3\tau_5} \\
=&\big(\tau_4 \times \tau_3 \times (\tau_3 \to \tau_1) \times (\tau_1 \to \tau_2) \to \tau_2\big) \times \big(\tau_5 \times \tau_3 \times (\tau_3 \to \tau_1) \times (\tau_1 \to \tau_2) \to \tau_2\big).
\end{aligned}
$$

Of course, some care needs to be taken when applying the rewrite rules, like first applying the associativity isomorphism and normalizing sub-expressions. To be precise, we provide a purely functional Agda implementation in Figure 1 and 2. This implementation also allows us to understand the restrictions imposed on types in normal form as it proves the following theorem.

Theorem 2.1. *If τ is a type in exp-log normal form, then $\tau \in$ D-Type, where*

$$
\begin{aligned}
\text{D-Type} \ni d, d' &::= c \mid c + d \\
\text{C-Type} \ni c, c' &::= a \mid a \times c \\
\text{Atom} \ni a &::= p \mid c \to p \mid c' \to c + d,
\end{aligned}
$$

and p is a type variable (base type).

```
module ENF (Proposition : Set) where

  infixr 6 _×_
  infixr 5 _+_
  infixr 4 _→_

  mutual
    data Atom : Set where
      _→ₚ_ : CNF ⟶ Proposition ⟶ Atom
      _→₊_+_ : CNF ⟶ CNF ⟶ ENF ⟶ Atom
      `_  : Proposition ⟶ Atom

    data CNF : Set where
      _×_ : Atom ⟶ CNF ⟶ CNF
      ᶜ_  : Atom ⟶ CNF

    data ENF : Set where
      _+_ : CNF ⟶ ENF ⟶ ENF
      ᵈ_  : CNF ⟶ ENF

  assoc× : CNF ⟶ CNF ⟶ CNF
  assoc× (ᶜ a) c′ = a × c′
  assoc× (a × c) c′ = a × (assoc× c c′)

  assoc+ : ENF ⟶ ENF ⟶ ENF
  assoc+ (ᵈ c) d′ = c + d′
  assoc+ (c + d) d′ = c + (assoc+ d d′)

  distrib₁ : CNF ⟶ ENF ⟶ ENF
  distrib₁ c (ᵈ c′) = ᵈ (assoc× c c′)
  distrib₁ c (c′ + d) = (assoc× c c′) + (distrib₁ c d)

  distrib : ENF ⟶ ENF ⟶ ENF
  distrib (ᵈ c) d′ = distrib₁ c d′
  distrib (c + d) d′ = assoc+ (distrib₁ c d′) (distrib d d′)

  explog₁ : CNF ⟶ ENF ⟶ CNF
  explog₁ c (c′ + d′) = ᶜ (c →₊ c′ + d′)
  explog₁ c (ᵈ (ᶜ (c′ →ₚ p′))) = ᶜ (assoc× c c′ →ₚ p′)
  explog₁ c (ᵈ (ᶜ (c′ →₊ c₁ + d₁))) = ᶜ (assoc× c c′ →₊ c₁ + d₁)
  explog₁ c (ᵈ (ᶜ (` p))) = ᶜ (c →ₚ  p)
  explog₁ c (ᵈ ((c′ →ₚ p′) × c′′)) =
    (assoc× c c′ →ₚ p′) × explog₁ c (ᵈ c′′)
  explog₁ c (ᵈ ((c′ →₊ c₁ + d₁) × c′′)) =
    (assoc× c c′ →₊ c₁ + d₁) × explog₁ c (ᵈ c′′)
  explog₁ c (ᵈ (` p × c′′)) = (c →ₚ p) × explog₁ c (ᵈ c′′)

  explog : ENF ⟶ ENF ⟶ CNF
  explog (ᵈ c) d′ = explog₁ c d′
  explog (c + d) d′ = assoc× (explog₁ c d′) (explog d d′)
```


Figure 1: Implementation of the type normalization function in Agda.

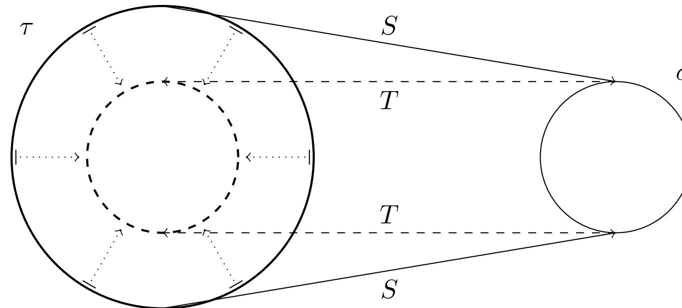
```
data Formula : Set where
  `_  : Proposition ⟶ Formula
  _+_ : Formula ⟶ Formula ⟶ Formula
  _×_ : Formula ⟶ Formula ⟶ Formula
  _⇒_ : Formula ⟶ Formula ⟶ Formula

enf : Formula ⟶ ENF
enf (` p) =  ᵈ (ᶜ (` p))
enf (f₁ + f₂) = assoc+ (enf f₁) (enf f₂)
enf (f₁ × f₂) = distrib (enf f₁) (enf f₂)
enf (f₁ ⇒ f₂) = ᵈ (explog (enf f₁) (enf f₂))
```

Figure 2: Implementation of the type normalization function in Agda.

From the inductive characterization of the previous theorem, it is immediate to notice that the exp-log normal form (ENF) is in fact an extension of the disjunctive normal form (DNF) to cover the function type. We shall now apply this simple and lossless transformation of types to the equational theory of terms of the lambda calculus with sums.

3 $\beta\eta$-Congruence classes at ENF type

Consider the lambda calculus from Figure 3. The virtue of type isomorphisms is that they preserve the equational theory of the inhabiting term calculus: an isomorphism between τ and σ is witnessed by a pair of lambda terms $T : \sigma \to \tau$ and $S : \tau \to \sigma$ such that $\lambda x.T(Sx) =_{\beta\eta} \lambda x.x$ and $\lambda y.S(Ty) =_{\beta\eta} \lambda y.y$. Therefore, if $\tau \cong \sigma$ and σ happens to be more canonical than τ — in the sense that to any $\beta\eta$-equivalence class of type τ corresponds a smaller one of type σ — one can reduce the problem of deciding $\beta\eta$-equality at τ to deciding it for a smaller subclass of terms.



In the case when $\sigma = \mathrm{enf}\,(\tau)$, σ is a priori more canonical than τ, since the effect of the reduction to ENF is to get rid of as many premises of sum types as possible,

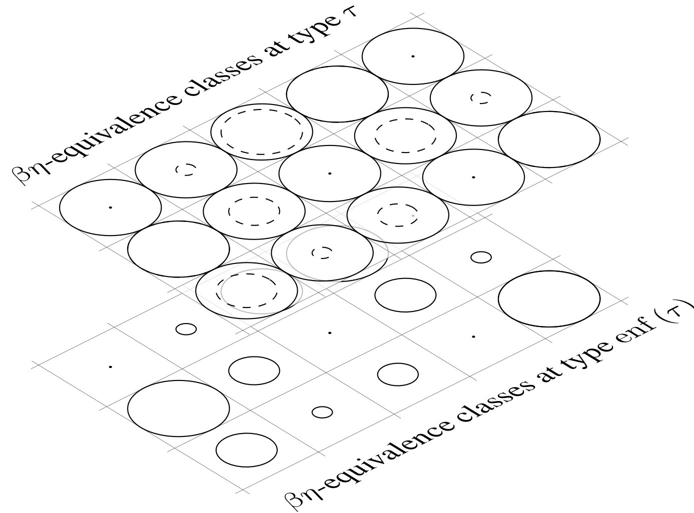
$$M, N ::= x^\tau \mid (M^{\tau\to\sigma} N^\tau)^\sigma \mid (\pi_1 M^{\tau\times\sigma})^\tau \mid (\pi_2 M^{\tau\times\sigma})^\sigma \mid \delta(M^{\tau+\sigma}, x_1^\tau.N_1^\rho, x_2^\sigma.N_2^\rho)^\rho \mid$$
$$\mid(\lambda x^\tau.M^\sigma)^{\tau\to\sigma} \mid \langle M^\tau, N^\sigma\rangle^{\tau\times\sigma} \mid (\iota_1 M^\tau)^{\tau+\sigma} \mid (\iota_2 M^\sigma)^{\tau+\sigma}$$

$$(\lambda x.N)M =_\beta N\{M/x\} \qquad (\beta_\to)$$
$$\pi_i\langle M_1, M_2\rangle =_\beta M_i \qquad (\beta_\times)$$
$$\delta(\iota_i M, x_1.N_1, x_2.N_2) =_\beta N_i\{M/x_i\} \qquad (\beta_+)$$
$$N =_\eta \lambda x.Nx \qquad x \notin \mathrm{FV}(N) \qquad (\eta_\to)$$
$$N =_\eta \langle \pi_1 N, \pi_2 N\rangle \qquad (\eta_\times)$$
$$N\{M/x\} =_\eta \delta(M, x_1.N\{\iota_1 x_1/x\}, x_2.N\{\iota_2 x_2/x\}) \qquad x_1, x_2 \notin \mathrm{FV}(N) \qquad (\eta_+)$$

Figure 3: Lambda terms and the equational theory $=_{\beta\eta}$.

and it is known that for the $\{\times, \to\}$-typed lambda calculus one can choose a single canonical η-long β-normal representative out of a class of $\beta\eta$-equal terms.

Thus, from the perspective of type isomorphisms, we can observe the partition of the set of terms of type τ into $=_{\beta\eta}$-congruence classes as projected upon different parallel planes in three dimensional space, one plane for each type isomorphic to τ. If we choose to observe the planes for τ and $\mathrm{enf}\,(\tau)$, we may describe the situation by the following figure.



The dashed circle depicts the compaction, if any, of a congruence class achieved by coercing to ENF type.

We do not claim that the plane of $\mathrm{enf}\,(\tau)$ is always the best possible plane to choose for deciding $=_{\beta\eta}$. Indeed, for concrete base types (think of the role of the unit type 1 in $(1 \to \tau + \sigma) \to \rho$) there may well be further isomorphisms to apply and hence a better plane than the one for $\mathrm{enf}\,(\tau)$. But, for the case of types where the sum can be completely eliminated, such as the two examples of Section 1, the projection amounts to compacting the $\beta\eta$-congruence class to a single point, a canonical normal term of type $\mathrm{enf}\,(\tau)$. Assuming τ, σ, τ_i are base types, the canonical representatives for the two $\beta\eta$-congruence classes of Section 1 are

$$\langle \lambda x.(\pi_1(\pi_2 x))(\pi_1 x), \lambda x.(\pi_2(\pi_2 x))(\pi_1 x)\rangle$$

and

$$\langle \lambda x.(\pi_1 x)((\pi_1\pi_2 x)(\pi_1\pi_2\pi_2 x)), \lambda x.(\pi_1 x)((\pi_1\pi_2 x)(\pi_1\pi_2\pi_2 x))\rangle.$$

Note that, unlike [4], we do not need any sophisticated term analysis to derive a canonical form in this kind of cases.

The natural place to pick a canonical representative is thus the (pseudo) normal type. Moreover, beware that even if it may be tempting to map a canonical representative along isomorphic coercions back to the original type, the obtained representative may not be truly canonical since there is generally more than one way to specify the terms S and T that witness a type isomorphism.

Of course, not always can all sum types be eliminated by type isomorphism, and hence not always can a class be compacted to a single point in that way. Nevertheless, even in the case where there are still sums remaining in the type of a term, the ENF simplifies the set of applicable $=_{\beta\eta}$-axioms.

Namely, since there is no restriction on the type of N from the left-hand side of η_+ and the right-hand side of $\beta_\to$, these two axioms overlap among themselves and with the others. Also, it is not clear how to choose between $\eta_\to$ and η_+, or $\eta_\times$ and η_+, when performing η-expansion. These complications are the reason why it is not simple, if possible, to have a confluent and strongly normalizing rewrite system for lambda calculus with sums.

For terms of ENF type, the complications can be largely avoided, as follows.

1. Since the syntax does not allow lambda abstractions to take an argument of sum type, for no M can the right-hand side of $\beta_\to$ and the left-hand side of η_+ overlap.

2. The η_+-axiom can be restricted to N of base or sum type, in presence of special cases of η_+ that allow to permute λ-s and π_i-s into the branches of δ. This removes the ambiguity when deciding which η-axiom to apply.

We get a restricted set of equations, $=^{\mathrm{e}}_{\beta\eta}$, shown in Figure 4, which is still complete for proving full $\beta\eta$-equality as made precise in the following theorem.

Theorem 3.1. *Let P, Q be terms of type τ and let $S : \tau \to \mathrm{enf}\,(\tau)\,, T : \mathrm{enf}\,(\tau) \to \tau$ be a witnessing pair of terms for the isomorphism $\tau \cong \mathrm{enf}\,(\tau)$. Then, $P =_{\beta\eta} Q$ if and only if $SP =^{\mathrm{e}}_{\beta\eta} SQ$ and if and only if $T(SP) =_{\beta\eta} T(SQ)$.*

$$
\begin{aligned}
M, N ::= {} & x^d \mid (M^{c \to p} N^c)^p \mid (M^{c \to c'+d} N^c)^{c'+d} \mid (\pi_1 M^{a \times c})^a \mid (\pi_2 M^{a \times c})^c \mid \delta(M^{c+d}, x_1^c.N_1^{d'}, x_2^d.N_2^{d'})^{d'} \\
& \mid (\lambda x^c.M^p)^{c \to p} \mid (\lambda x^c.M^{c'+d})^{c \to c'+d} \mid \langle M^a, N^c \rangle^{a \times c} \mid (\iota_1 M^c)^{c+d} \mid (\iota_2 M^d)^{c+d}
\end{aligned}
$$

$$
\begin{aligned}
(\lambda x^{c'}.N^{\substack{p \\ c+d}})M &=^{\mathrm{e}}_{\beta} N\{M/x\} && (\beta^{\mathrm{e}}_{\to}) \\
\pi_i \langle M_1^a, M_2^c \rangle &=^{\mathrm{e}}_{\beta} M_i && (\beta^{\mathrm{e}}_{\times}) \\
\delta(\iota_i M, x_1.N_1, x_2.N_2)^d &=^{\mathrm{e}}_{\beta} N_i\{M/x_i\} && (\beta^{\mathrm{e}}_{+}) \\
N^{\substack{c \to p \\ c' \to c+d}} &=^{\mathrm{e}}_{\eta} \lambda x.Nx && x \notin \mathrm{FV}(N) \quad (\eta^{\mathrm{e}}_{\to}) \\
N^{a \times c} &=^{\mathrm{e}}_{\eta} \langle \pi_1 N, \pi_2 N \rangle && (\eta^{\mathrm{e}}_{\times}) \\
N^{\substack{p \\ c+d}}\{M/x\} &=^{\mathrm{e}}_{\eta} \delta(M, x_1.N\{\iota_1 x_1/x\}, x_2.N\{\iota_2 x_2/x\}) && x_1, x_2 \notin \mathrm{FV}(N) \quad (\eta^{\mathrm{e}}_{+}) \\
\pi_i \delta(M, x_1.N_1, x_2.N_2) &=^{\mathrm{e}}_{\eta} \delta(M, x_1.\pi_i N_1, x_2.\pi_i N_2)^c && (\eta^{\mathrm{e}}_{\pi}) \\
\lambda y.\delta(M, x_1.N_1, x_2.N_2) &=^{\mathrm{e}}_{\eta} \delta(M, x_1.\lambda y.N_1, x_2.\lambda y.N_2)^{\substack{c \to p \\ c' \to c+d}} && y \notin \mathrm{FV}(M) \quad (\eta^{\mathrm{e}}_{\lambda})
\end{aligned}
$$

Figure 4: Lambda terms of ENF type and the equational theory $=^{\mathrm{e}}_{\beta\eta}$.

Proof. Since the set of terms of ENF type is a subset of all typable terms, it suffices to show that all $=_{\beta\eta}$-equations that apply to terms at ENF type can be derived already by the $=^{\mathrm{e}}_{\beta\eta}$-equations.

Notice first that $\eta^{\mathrm{e}}_{\lambda}$ and η^{e}_{π} are special cases of η_+, so, in fact, the only axiom missing from $=^{\mathrm{e}}_{\beta\eta}$ is η_+ itself,

$$
N\{M/x\} =^{\mathrm{e}}_{\eta} \delta(M, x_1.N\{\iota_1 x_1/x\}, x_2.N\{\iota_2 x_2/x\}) \qquad (x_1, x_2 \notin \mathrm{FV}(N)),
$$

when N is of type $c \to p$, $c' \to c + d$, or $a \times c$. We show that this axiom is derivable from the $=^{\mathrm{e}}_{\beta\eta}$-ones, for N^d, by induction on d.

Case for N of type $a \times c$.

$$
\begin{aligned}
& N\{M/x\} \\
=^{\mathrm{e}}_{\eta} & \langle \pi_1(N\{M/x\}), \pi_2(N\{M/x\})\rangle && \text{by } \eta^{\mathrm{e}}_{\times} \\
= & \langle (\pi_1 N)\{M/x\}, (\pi_2 N)\{M/x\}\rangle \\
=^{\mathrm{e}}_{\eta} & \langle \delta(M, x_1.(\pi_1 N)\{\iota_1 x_1/x\}, x_2.(\pi_1 N)\{\iota_2 x_2/x\}), \\
& \quad \delta(M, x_1.(\pi_2 N)\{\iota_1 x_1/x\}, x_2.(\pi_2 N)\{\iota_2 x_2/x\})\rangle && \text{by ind. hyp.} \\
=^{\mathrm{e}}_{\beta\eta} & \langle \pi_1(\delta(M, x_1.N\{\iota_1 x_1/x\}, x_2.N\{\iota_2 x_2/x\})), \\
& \quad \pi_2(\delta(M, x_1.N\{\iota_1 x_1/x\}, x_2.N\{\iota_2 x_2/x\}))\rangle && \text{by } \eta^{\mathrm{e}}_{\pi} \\
=^{\mathrm{e}}_{\eta} & \delta(M, x_1.N\{\iota_1 x_1/x\}, x_2.N\{\iota_2 x_2/x\}) && \text{by } \eta^{\mathrm{e}}_{\times}
\end{aligned}
$$

Case for N of type $c \to p$.

$$
\begin{aligned}
& N\{M/x\} \\
=^{\mathrm{e}}_{\eta} & \lambda y.(N\{M/x\})y && \text{by } \eta^{\mathrm{e}}_{\to} \\
= & \lambda y.(Ny)\{M/x\} && \text{for } y \notin \mathrm{FV}\,(N\{M/x\}) \\
=^{\mathrm{e}}_{\eta} & \lambda y.\delta(M, x_1.(Ny)\{\iota_1 x_1/x\}, x_2.(Ny)\{\iota_2 x_2/x\}) && \text{by } \eta^{\mathrm{e}}_{+} \text{ for } p \\
=^{\mathrm{e}}_{\eta} & \delta(M, x_1.(\lambda y.Ny)\{\iota_1 x_1/x\}, x_2.(\lambda y.Ny)\{\iota_2 x_2/x\}) && \text{by } \eta^{\mathrm{e}}_{\lambda} \\
=^{\mathrm{e}}_{\eta} & \delta(M, x_1.N\{\iota_1 x_1/x\}, x_2.N\{\iota_2 x_2/x\}) && \text{by } \eta^{\mathrm{e}}_{\to}
\end{aligned}
$$

Case for N of type $c' \to c + d$. The proof is the same as for the previous case but η^{e}_{+} is applied for $c + d$ instead of p.

□

The transformation of terms to ENF type thus allows to disentangle the axioms of $=_{\beta\eta}$. One could get rid of η^{e}_{π} and $\eta^{\mathrm{e}}_{\lambda}$ if one had a version of λ-calculus resistant to these permuting conversions. The syntax of such a lambda calculus would further be simplified if, instead of binary, one had n-ary sums and products. In that case, there would be no need for variables of sum type at all (currently they can only be introduced by the second branch of δ). We would in fact get a pattern-matching calculus, with only variables of type a, and that would still be suitable as a small theoretical core of functional programming languages.

4 Conclusion

We have described how to systematically remove sum hypotheses from a type while preserving isomorphism. The type normal form obtained is so simple that it is surprising it has not been isolated before. We also showed how it can be applied for

disentangling the axioms of the standard $\beta\eta$-equality for terms of the lambda calculus with sums.

The idea to apply type isomorphism for deciding equality of terms appears to be new, although it has been implicitly used before [1, 17]. Namely, in the so called focusing approach from sequent calculi, one gets a more canonical representation of terms (proofs) by grouping all so called asynchronous proof rules into blocks. However, while all asynchronous proof rules are special kinds of type isomorphisms, not all type isomorphisms are accounted by the asynchronous rules. Our approach can thus been seen as generalizing the focusing methodology.

The literature on studying the theory of $\beta\eta$-equality by a syntax-oriented analysis of terms is quite rich [9, 10, 12, 2, 8, 4, 15]. In that context, our work can be seen as an orthogonal contribution, since one could only profit from the additional syntactic restrictions imposed by type isomorphisms.

The exp-log normal form could potentially be useful for automated theorem proving for intuitionistic logic, because it enlarges the phase at which no backtracking is necessary during proof search, but also for inductive theorem proving in presence of exponential polynomials, as well as contexts where terms representations modulo type isomorphism is important like Homotopy Type Theory.

Acknowledgments

This paper was financed by ERC Advanced Grant ProofCert. I would like to thank Zakaria Chihani for providing enthusiastic feedback, Anupam Das for mentioning exponential fields, and Gabriel Scherer and Beniamino Accattolli for giving valuable advice regarding evaluation contexts.

References

[1] Arbob Ahmad, Dan Licata, and Robert Harper. Deciding coproduct equality with focusing. Manuscript, 2010.

[2] T. Altenkirch, P. Dybjer, M. Hofmann, and P. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *Logic in Computer Science, 2001. Proceedings. 16th Annual IEEE Symposium on*, pages 303–310, 2001.

[3] Vincent Balat. Keeping sums under control. In *Workshop on Normalization by Evaluation*, pages 11–20, Los Angeles, United States, August 2009.

[4] Vincent Balat, Roberto Di Cosmo, and Marcelo Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 64–76, New York, NY, USA, 2004. ACM.

[5] Kim B. Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2:231–247, 6 1992.

[6] Stanley Burris and Simon Lee. Tarski's high school identities. *American Mathematical Monthly*, 100:231–236, 1993.

[7] Stanley N. Burris and Karen A. Yeats. The saga of the high school identities. *Algebra Universalis*, 52:325–342, 2004.

[8] Roberto Di Cosmo and Delia Kesner. A confluent reduction for the extensional typed λ-calculus with pairs, sums, recursion and terminal object. In Andrzej Lingas, Rolf Karlsson, and Svante Carlsson, editors, *Automata, Languages and Programming*, volume 700 of *Lecture Notes in Computer Science*, pages 645–656. Springer Berlin Heidelberg, 1993.

[9] Daniel Dougherty. Some lambda calculi with categorical sums and products. In *Rewriting Techniques and Applications*, pages 137–151. Springer, 1993.

[10] Daniel J. Dougherty and Ramesh Subrahmanyam. Equality between functionals in the presence of coproducts. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science*, LICS '95, pages 282–, Washington, DC, USA, 1995. IEEE Computer Society.

[11] Marcelo 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*, 141:35–50, 2006.

[12] Neil Ghani. $\beta\eta$-equality for coproducts. In *Typed Lambda Calculi and Applications*, pages 171–185. Springer, 1995.

[13] Godfrey Harold Hardy. *Orders of Infinity. The 'Infinitärcalcül' of Paul Du Bois-Reymond*. Cambridge Tracts in Mathematic and Mathematical Physics. Cambridge University Press, 1910.

[14] 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)*, CSL-LICS '14, pages 53:1–53:7, New York, NY, USA, 2014. ACM.

[15] Sam Lindley. Extensional rewriting with sums. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications*, volume 4583 of *Lecture Notes in Computer Science*, pages 255–271. Springer Berlin Heidelberg, 2007.

[16] Guillaume Munch-Maccagnoni and Gabriel Scherer. Polarised Intermediate Representation of Lambda Calculus with Sums. In *Proceedings of the Thirtieth Annual ACM/IEEE Symposium on Logic In Computer Science (LICS 2015)*, 2015. To appear.

[17] Gabriel Scherer. Multi-focusing on extensional rewriting with sums. In *Proceedings of TLCA 2015*, 2015. To appear.