On the exp-log normal form of types
Danko Ilik

To cite this version:

HAL Id: hal-01167162
https://inria.hal.science/hal-01167162v2
Submitted on 17 Aug 2016

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.

Copyright

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.
The exp-log normal form of types
Decomposing extensional equality and representing terms compactly

Danko Ilik
Inria & LIX, Ecole Polytechnique
91128 Palaiseau Cedex, France
danko.ilik@inria.fr

Abstract
Lambda calculi with algebraic data types lie at the core of functional programming languages and proof assistants, but conceal at least two fundamental theoretical problems already in the presence of the simplest non-trivial data type, the sum type. First, we do not know of an explicit and implemented algorithm for deciding the beta-eta-equiality of terms—and this in spite of the first decidability results proven two decades ago. Second, it is not clear how to decide when two types are essentially the same, i.e. isomorphic, in spite of the meta-theoretic results on decidability of the isomorphism.

In this paper, we present the exp-log normal form of types—derived from the representation of exponential polynomials via the unary exponential and logarithmic functions—that any type built from arrows, products, and sums, can be isomorphically mapped to. The type normal form can be used as a simple heuristic for deciding type isomorphism, thanks to the fact that it is a systematic application of the high-school identities.

We then show that the type normal form allows to reduce the standard beta-eta equational theory of the lambda calculus to a representation 

\[ \lambda x.\delta(x, z.t_1 z, z.t_2 z) \]

of the simplest non-trivial data type, the sum type. First, we do not know of an explicit and implemented algorithm for deciding type isomorphism, thanks to the fact that it is a systematic application of the high-school identities.

We end by describing an alternative representation of normal terms of the lambda calculus with sums, together with a Coq-implemented converter into/from our new term calculus. The difference with the only other previously implemented heuristic for deciding interesting instances of eta-equality by Balat, Di Cosmo, and Fiore, is that we exploit the type information of terms substantially and this often allows us to obtain a canonical representation of terms without performing sophisticated term analyses.

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 directly. A natural model for the core of functional languages should at least include direct support for a simplest case of variant types, \textit{sums}, and of records i.e. \textit{product} types. But, unlike the theory of the \( \rightarrow \)-typed lambda calculus, the theory of the \( \{\rightarrow, +, \times\} \)-typed one is not all roses.

\textbf{Canonicity of normal terms and \( \eta \)-equality} A first problem is canonicity of normal forms of terms. Take, for instance, the term \( \lambda x.\lambda y.\delta(x, z.t_1 z, z.t_2 z) \), and three of its \( \eta \)-long representations,

\[ \lambda x.\lambda y.\delta(x, z.t_1 z, z.t_2 z) \]
\[ \lambda x.\lambda y.(x, z.t_1 z, z.t_2 z) \]
\[ \lambda x.\delta(x, z.t_1 z, z.t_2 z) \]

where \( \delta \) is a pattern matching construct, i.e. a \textit{case-expression} analysing the first argument, with branches of the pattern matching given via the variable \( z \) in the second and third argument.

These three terms are all equal with respect to the standard equational theory \( \beta\eta \) of the lambda calculus (Figure 1), but why should we prefer any one of them over the others to be a \textit{canonical} representative of the class of equal terms?

Or, consider the following two terms of type \( (\tau_1 \rightarrow \tau_2) \rightarrow (\tau_1 \rightarrow \tau_3) \rightarrow (\tau_1 + \tau_3) \rightarrow \tau_2 \) (example taken from [Balat et al. 2004]):

\[ \lambda xyz.t.u(x(yz)) \]
\[ \lambda t.u.\delta(\delta(u, x.t_1 z, z.t_2 z), y_1.x(y_1), y_2.x.y_2) \]

These terms are \( \beta\eta \)-equal, but can one easily notice the equality? In order to do so, since both terms are \( \beta \)-normal, one would need to do non-trivial \( \beta \)- and \( \eta \)-expansions (see Example 3 in Section 4).

The lambda calculus over the restricted language of types—when the sum type is absent—these problems do not exist, since \( \beta \)-normalization followed by an \( \eta \)-expansion is deterministic and produces a canonical representative for any class of \( \beta\eta \)-equal terms. Deciding \( =_{\beta\eta} \) for that restricted calculus amounts to comparing canonical forms up to syntactic identity.

In the presence of sums, we only have a notion of canonical \textit{interpretation} of terms in the category of sheaves for the Grothendieck topology over the category of constrained environments [Altenkirch et al. 2001], as well as the sophisticated normal form of terms due to Balat, Di Cosmo, and Fiore which is not canonical (unique) syntactically (Balat et al. 2004). Balat et al. also...
provide an implementation of a type-directed partial evaluator that normalizes terms to their normal form, and this represented up to now the only implemented heuristic for deciding $\beta\eta$-equality—it is not a full decision procedure, because the normal forms are not canonical. We shall discuss these and the other decidability results some more in Related work of Section 5.

Treating full $\beta\eta$-equality is hard, even if, in practice, we often only need to treat special cases of it, such as certain commuting conversions.

**Recognizing isomorphic types** If we leave aside the problems of canonicity of and equality between terms, there is a further problem at the level of types that makes it hard to determine whether two type signatures are essentially the same one. Namely, although for each of the type languages $\{\to, \times\}$ and $\{\to, +\}$ there is a very simple algorithm for deciding type isomorphism, for the whole of the language $\{\to, +,\times\}$ it is only known that type isomorphism is decidable when types are to be interpreted as finite structures, and that without a practically implementable algorithm in sight (Ilik 2014). Although the notion of isomorphism can be treated abstractly in Category Theory, in bi-Cartesian closed categories, 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 1), the types $\tau$ and $\sigma$ are isomorphic when there exist coercing lambda terms $M : \sigma \to \tau$ and $N : \tau \to \sigma$ such that

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

In other words, data/programs can be converted back and forth between $\tau$ and $\sigma$ without loss of information.

The problem of isomorphism is in fact closely related to the famous Tarski High School Identities Problem (Burris and Yeats 2004; Fiore and et al. 2006). What is important for us here is that types can be seen as just arithmetic expressions: if the type $\tau \to \sigma$ is denoted by the binary arithmetic exponentiation $\sigma^\tau$, then every type $\rho$ denotes at the same time an exponential polynomial $p$. 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 $\sigma^n$ for a concrete $n \in \mathbb{N}$.

Moreover, we have that

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

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

This hence provides an algorithm for deciding non-isomorphism: given 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:

$\tau \cong \sigma$ if and only if $\mathbb{N}^+ \models \tau = \sigma$.

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 canonical normal form obtained by type transformation that follows currying (Rittri 1991).

Given that it is not known whether one can find such a canonical normal form for the full language of types (Ilik 2014), what we can hope to do in practice is to find at least a pseudo-canonical normal form. We shall now define such a type normal form.

**The exp-log normal form of types**

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),

$\tau \equiv_\beta \sigma ::= \chi_i \mid \tau \to \sigma \mid \tau \times \sigma \mid \tau + \sigma$,

where $\chi_i$ are atomic types (or type variables), it is not always clear when two given types are essentially the same one. More precisely, it is not known how to decide whether two types are isomorphic (Ilik 2014). What is important for us here is that types can be seen as just arithmetic expressions: if the type $\tau \to \sigma$ is denoted by the binary arithmetic exponentiation $\sigma^\tau$, then every type $\rho$ denotes at the same time an exponential polynomial $p$. 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 $\sigma^n$ for a concrete $n \in \mathbb{N}$.

Moreover, we have that

$\tau \cong \sigma$ implies $\mathbb{N}^+ \models \tau = \sigma$.

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 canonical normal form obtained by type transformation that follows currying (Rittri 1991).

Given that it is not known whether one can find such a canonical normal form for the full language of types (Ilik 2014), what we can hope to do in practice is to find at least a pseudo-canonical normal form. We shall now define such a type normal form.

The idea is to use the decomposition of the binary exponential function $\sigma^\tau$ 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

$\sigma^\tau = e^{\tau \times \log \sigma}$ \quad 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 (Hardy 1910) 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
\exp(\tau_1 \times \tau_2) = \exp(\tau_1) \times \exp(\tau_2)
log(\tau_1 \times \tau_2) = \log(\tau_1) + \log(\tau_2)

In other words, \exp and log can be considered as macro expansions rather than unary type constructors. Let us take the type \(\tau, \sigma, \rho\) are atomic types. It can be normalized in the following way:
\begin{align*}
\tau + \sigma &\to (\tau + \sigma \to \rho) \to \rho = \\
&= (\rho^{\tau + \sigma})^{\tau + \sigma} = \\
&= \exp((\tau + \sigma) \log[\exp((\tau + \sigma) \log \rho) \log \rho]) \leadsto \\
&\leadsto \exp((\tau + \sigma) \log[\exp(\tau \log \rho) \exp(\sigma \log \rho) \log \rho]) \leadsto \\
&\leadsto \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 \sigma \rho} \rho^{\tau \rho} \rho \rho^{\sigma \rho} = \\
&= (\tau \times (\tau \to \rho) \times (\sigma \to \rho) \to \rho) \times (\sigma \times (\tau \to \rho) \times (\sigma \to \rho) \to \rho).
\end{align*}

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). Be believe the link to abstract algebra is well work keeping in mind, since it may give rise to further cross-fertilization between mathematics and the theory of programming languages. However, from the operational point of view, all this transformation does is that it prioritized and orients the high-school identities,
\begin{align*}
(f + g) + h &\leadsto f + (g + h) \quad (1) \\
(fg)h &\leadsto f(gh) \quad (2) \\
f(g + h) &\leadsto fg + fh \quad (3) \\
(f + g)h &\leadsto fh + gh \quad (4) \\
f^{g + h} &\leadsto f^g f^h \quad (5) \\
(fg)^h &\leadsto f^h g^h \quad (6) \\
(f^g)^h &\leadsto f^{hg} \quad (7)
\end{align*}
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{align*}
(\tau_1 \to \tau_2) \to (\tau_1 \to \tau_1) \to \tau_1 \to \tau_1 + \tau_0 &\to \tau_2 = \\
&= ((\tau_2^{\tau_1 + \tau_0} \tau_1 \tau_1) \tau_2 \tau_1) \leadsto \\
&\leadsto \tau_2^{\tau_1 + \tau_1 + \tau_0} \tau_1 \tau_1 \tau_2 \tau_1 \tau_1 \tau_2 \tau_1 \tau_1 \tau_1 \tau_0 = \\
&= (\tau_4 \times \tau_3 \times (\tau_2 \tau_1) \times (\tau_1 \to \tau_2) \to \tau_2) \times \\
&\times (\tau_5 \times \tau_3 \times (\tau_3 \to \tau_1) \times (\tau_1 \to \tau_2) \to \tau_2).
\end{align*}

Of course, some care needs to be taken when applying the rewrite rules, in order for the procedure to be deterministic, like giving precedence to the type rewrite rules and normalizing subexpressions. To be precise, we provide a purely functional Coq implementation below. This is just one possible implementation of the rewriting rules, but being purely functional and structurally recursive (i.e. terminating) it allows us to understand the restrictions imposed on types in normal form, as it proves the following theorem.

**Theorem 1.** If \(\tau\) is a type in exp-log normal form, then \(\tau\in ENF, where
\end{align*}

\begin{align*}
\text{ENF} \ni e &\leadsto c | d,
\end{align*}

where
\begin{align*}
\text{DNF} \ni d, d_i &\ni (c_1 + (c_2 + \cdots + n)) \cdots n \geq 2 \\
\text{CNF} \ni c, c_i &\ni (c_1 \to b_1) \times \cdots \times (c_n \to b_n) \cdots n \geq 0 \\
\text{Base} \ni b, b_i &\ni p | d,
\end{align*}

and \(p\) denotes atomic types (type variables).

Assuming a given set of atomic types,

\begin{align*}
\text{Parameter Proposition : Set},
\end{align*}

the goal is to map the unrestricted language of types, given by the inductive definition\(^1\)

\begin{align*}
\text{Inductive Formula : Set :=}
| \text{prop : Proposition \to Formula} \\
| \text{disj : Formula \to Formula \to Formula} \\
| \text{conj : Formula \to Formula \to Formula}
\end{align*}

\(^1\) May the reader to forgive us for the implicit use of the Curry-Howard correspondence in the Coq code snippets, where we refer to types and type constructors as formulas and formula constructors.
into the exp-log normal form which fits in the following inductive
signature.

\[\text{Inductive CNF : Set :=}
\begin{align*}
&\text{top} \\
&\text{con : CNF → Base → CNF → CNF} \\
&\text{with DNF : Set :=} \\
&\text{two : CNF → CNF → DNF} \\
&\text{dis : CNF → DNF → DNF} \\
&\text{with Base : Set :=} \\
&\text{prp : Proposition → Base} \\
&\text{bd : DNF → Base.}
\end{align*}\]

\[\text{Inductive ENF : Set :=}
\begin{align*}
&\text{cnf : CNF → ENF} \\
&\text{dnf : DNF → ENF.}
\end{align*}\]

The \(\text{con c1 b c2}\) constructor corresponds to \(b^{c1}c2\) or the type
\((c1 → b) × c2\) from Theorem 1. The normalization function, \(\text{enf ()}\),
is defined using the following fixpoints:

\(\text{nplus}\) which makes a flattened \(n\)-ary sum out of two given \(n\)-ary
sums, i.e. implements the \(+\)-associativity rewriting (1).

\(\text{ntimes}\) which is analogous to ‘\(nplus\)’, but for products, implement-
ing (2).

\(\text{distr}\) which performs the distributivity rewriting, (3) and (4), and
\(\text{explogn}\) which performs the rewriting involving exponentiations,
(5), (6), and (7).

\[\text{Fixpoint explogn (c : CNF)(e : ENF) : CNF :=}
\begin{align*}
&\text{match e with} \\
&\text{cnf a ⇒ explogn (enf c e1) e2} \\
&\text{dnf d1 ⇒ explogn (enf c e1) e2} \end{align*}\]

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

3. \(\beta\eta\)-Congruence classes at ENF type

The virtue of type isomorphisms is that they preserve the equational theory of the term calculus: an isomorphism between \(\tau\) and \(\sigma\) is witnessed by a pair of lambda terms

\[ T : \sigma \rightarrow \tau \quad \text{and} \quad S : \tau \rightarrow \sigma \]

such that

\[ \lambda x. T(Sx) =_{\beta\eta} \lambda x. x \quad \text{and} \quad \lambda y. T(Ty) =_{\beta\eta} \lambda y. y. \]

Therefore, when \(\tau \cong \sigma\), and \(\sigma\) happens to be more canonical than \(\tau\)—in the sense that to any \(\beta\eta\)-equivalence class of type \(\tau\) corresponds a smaller one of type \(\sigma\)—one can reduce the problem of deciding \(\beta\eta\)-equality at \(\tau\) to deciding it for a smaller subclass of terms.

In the case when \(\sigma = \text{enf}(\tau)\), the equivalence classes at type \(\sigma\) will not be larger than their original classes at \(\tau\), since the main effect of the reduction to exp-log normal form is to get rid of as many sum types on the left of an arrow as possible, and it is known that for the \(\{x, \rightarrow\}\)-typed lambda calculus one can choose a single canonical \(\eta\)-long \(\beta\)-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 \(\tau\) into \(\equiv_{\beta\eta}\)-congruence classes as projected upon different parallel planes in three dimensional space, one plane for each type isomorphic to \(\tau\). If we choose to observe the planes for \(\tau\) and \(\text{enf}(\tau)\), we may describe the situation by the following figure.

![Diagram of type \(\equiv_{\beta\eta}\) classes](image)

The dashed circle depicts the compaction, if any, of a congruence class achieved by coercing to ENF type. The single point depicts the compaction to a singleton set, the case where a unique canonical representative of a class of \(\beta\eta\)-terms exists.

We do not claim that the plane of \(\text{enf}(\tau)\) is always the best possible plane to choose for deciding \(\equiv_{\beta\eta}\). Indeed, for concrete base types there may well be further type isomorphisms to apply (think of the role of the unit type 1 in \((1 \rightarrow \tau + \sigma) \rightarrow \rho\) and hence a better plane than the one for \(\text{enf}(\tau)\). However, it is a reasonably good default choice.

For the cases 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 \(\text{enf}(\tau)\).

Assuming \(\tau, \sigma, \tau_1\) are base types, the canonical representatives for the two \(\beta\eta\)-congruence classes of Section 1 are

\[ \langle \lambda x. (\pi_1(\pi_2x)) \rangle(\pi_1x) =_{\beta\eta} (\lambda x. (\pi_2(\pi_2x)))(\pi_1x) \]

and

\[ \langle \lambda x. (\pi_1x)(\pi_1(\pi_2x)) \rangle(\pi_1x) =_{\beta\eta} (\lambda x. (\pi_2(\pi_2x))(\pi_1x)\pi_2x) \rangle(\pi_1x) \]

Note that, unlike (Balat et al. 2004), we do not need any sophisticated term analysis to derive a canonical form in this kind of cases. One may either apply the standard terms witnessing the isomorphisms by hand, or use our normalizer described in Section 4.

The natural place to pick a canonical representative is thus the \(\beta\eta\)-congruence class of terms at the normal type, not the class at the original 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 \(\equiv_{\beta\eta}\)-axioms.

We can use it to get a restricted set of equations, \(\equiv_{\beta\eta}\), shown in Figure 2 which is still complete for proving full \(\beta\eta\)-equality, as made precise in the following theorem.

**Theorem 2.** Let \(P, Q\) be terms of type \(\tau\) and let \(S : \tau \rightarrow \text{enf}(\tau), T : \text{enf}(\tau) \rightarrow \tau\) be a witnessing pair of terms for the isomorphism \(\tau \cong \text{enf}(\tau)\). Then, \(\equiv_{\beta\eta}\) if and only if \(ST = SQ\) and if and only if \(T(SP) = SQ\).

Proof. Since the set of terms of ENF type is a subset of all typable terms, it suffices to show that all \(\equiv_{\beta\eta}\)-equations that apply to terms of ENF type can be derived already by the \(\equiv_{\beta\eta}\)-equations.

Notice first that \(\equiv^\omega_{\beta\eta}\) and \(\equiv_p\) are special cases of \(\equiv_{\beta\eta}\), so, in fact, the only axiom missing from \(\equiv_{\beta\eta}\) is \(\equiv_p\) itself.

\[ N\{M/x\} =_{\beta\eta} \delta(M, x_1, N\{x_1/x_1\}, x_2, N\{x_2/x_2\}) \]

\[ (x_1, x_2 \notin \text{FV}(N)) \]

when \(N\) is of type \(c\); the case of \(N\) of type \(d\) is covered directly by the \(\equiv_p\) axiom. We thus show that the \(\equiv_p\) axiom is derivable from the \(\equiv_{\beta\eta}\)-ones by induction on \(c\).

**Case for \(N\) of type \((c \rightarrow b) \times c_0\).**

\[ N\{M/x\} =_{\beta\eta}^\omega \delta(M, x_1, N\{x_1/x_1\}, x_2, N\{x_2/x_2\}) \]

\[ \text{by } \equiv_p^\omega \]

\[ =_{\beta\eta}^\omega (\{x_1, \delta(M, x_1, \pi_1N\{x_1/x_1\}, x_2, \pi_2N\{x_2/x_2\})\}) \]

\[ =_{\beta\eta}^\omega (\{x_1, \pi_1(N\{x_1/x_1\}, x_2, N\{x_2/x_2\})\}) \]

\[ =_{\beta\eta}^\omega \delta(M, x_1, \pi_1\{x_1/x_1\}, x_2, N\{x_2/x_2\}) \]

\[ \text{by } \equiv_p^\omega \]

\[ =_{\beta\eta}^\omega \delta(M, x_1, N\{x_1/x_1\}, x_2, N\{x_2/x_2\}) \]

\[ \text{by } \equiv_p^\omega \]
The transformation of terms to ENF type thus allows to simplify the (up to now) standard axioms of $\equiv_{\beta\eta}$. The new axioms are complete for $\equiv_{\beta\eta}$ in spite of them being only special cases of the old ones. A notable feature is that we get to disentangle the left-hand side and right-hand side of the equality axioms: for instance, the right-hand side of \( N(x) \) axiom can no longer overlap with the left-hand side of the $\equiv_{\beta\eta}$ axiom, by typing restrictions on the term $M$.

One could get rid of $\equiv_{\beta\eta}$ and $\equiv_{\eta}$ if one had a version of $\lambda$-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 $\delta$). We would in fact get a calculus with only variables of type $c \rightarrow b$, and that would still be suitable as a small theoretical core of functional programming languages.

4. A compact representation of terms at ENF type

It is the subject of this section to show that the desiderata for a more canonical calculus from the previous paragraph can in fact be achieved. We shall define a new representation of lambda terms, that we have isolated as the most compact syntax possible during the formal Coq development of a normalizer of terms at ENF type. The description of the normalizer itself will be left for the second part of this section. Subsection 4.1. In the first part of the section, we shall demonstrate the value of representing terms in our calculus on a number of examples. Comparing our normal form for syntactical identity provides a first such heuristic for deciding $\equiv_{\beta\eta}$ in the presence of sums.

Before we continue with the presentation of the new calculus, for the sake of precision, we give the formal representation of terms of the two term calculi. First, we represent the usual lambda calculus with sums.

<table>
<thead>
<tr>
<th>Inductive HSc : CNF → Set :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{tt} : \text{HSc}\ top$</td>
</tr>
<tr>
<td>$\text{pair} : \forall {c1 \ b\ c2}, \text{HSc}\ c1\ b \rightarrow \text{HSc}\ c2 \rightarrow \text{HSc}\ (\text{con}\ c1\ b\ c2)$</td>
</tr>
</tbody>
</table>

The constructors are self-explanatory, except for $\text{hyp}$ and $\text{wkn}$, which are in fact used to denote de Bruijn indices: $\text{hyp}$ denotes 0, while $\text{wkn}$ is the successor. For instance, the term $\lambda x y z . y$ is represented as $\text{lam}\ (\text{lam}\ (\text{lam}\ \text{wkn}))$ i.e. $\text{lam}\ (\text{lam}\ \text{lam})$.

DeBruijn indices creep in as the simplest way to work with binders in Coq, and although they may reduce readability, they solve the problem with $\alpha$-conversion of terms.

Next is our compact representation of terms, defined by the following simultaneous inductive definition of terms at base type (HSc), together with terms at product type (HSc). These later are simply finite lists of HSc-terms.

<table>
<thead>
<tr>
<th>Inductive ND : list Formula → Formula → Set :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{hyp} : \forall {\text{Gamma} A}$</td>
</tr>
<tr>
<td>$\text{ND} (A :: \text{Gamma} A)$</td>
</tr>
<tr>
<td>$\text{wkn} : \forall {\text{Gamma} A B}$</td>
</tr>
<tr>
<td>$\text{ND Gamma} A \rightarrow \text{ND} (B :: \text{Gamma} A)$</td>
</tr>
<tr>
<td>$\text{lam} : \forall {\text{Gamma} A B}$</td>
</tr>
<tr>
<td>$\text{ND Gamma} (\text{impl} A B)$</td>
</tr>
<tr>
<td>$\text{app} : \forall {\text{Gamma} A B}$</td>
</tr>
<tr>
<td>$\text{ND Gamma} A \rightarrow \text{ND Gamma} B$</td>
</tr>
<tr>
<td>$\text{pair} : \forall {\text{Gamma} A B}$</td>
</tr>
<tr>
<td>$\text{ND Gamma} (\text{conj} A B)$</td>
</tr>
<tr>
<td>$\text{fst} : \forall {\text{Gamma} A B}$</td>
</tr>
<tr>
<td>$\text{ND Gamma} (\text{disj} A B)$</td>
</tr>
<tr>
<td>$\text{snd} : \forall {\text{Gamma} A B}$</td>
</tr>
<tr>
<td>$\text{ND Gamma} (\text{disj} A B)$</td>
</tr>
<tr>
<td>$\text{inf} : \forall {\text{Gamma} A B}$</td>
</tr>
<tr>
<td>$\text{ND Gamma} (\text{con} A B)$</td>
</tr>
<tr>
<td>$\text{cas} : \forall {\text{Gamma} A B C}$</td>
</tr>
<tr>
<td>$\text{ND Gamma} (\text{disj} A B)$</td>
</tr>
</tbody>
</table>

The constructors are self-explanatory, except for $\text{hyp}$ and $\text{wkn}$, which are in fact used to denote de Bruijn indices: $\text{hyp}$ denotes 0, while $\text{wkn}$ is the successor. For instance, the term $\lambda x y z . y$ is represented as $\text{lam}\ (\text{lam}\ (\text{lam}\ \text{wkn}))$ i.e. $\text{lam}\ (\text{lam}\ \text{lam})$.

DeBruijn indices creep in as the simplest way to work with binders in Coq, and although they may reduce readability, they solve the problem with $\alpha$-conversion of terms.

Next is our compact representation of terms, defined by the following simultaneous inductive definition of terms at base type (HSc), together with terms at product type (HSc). These later are simply finite lists of HSc-terms.
For a more human-readable notation of our calculus, we are going to use the following one.

\[ P, Q :::= (M_1, \ldots, M_n) \quad (n \geq 0) \]

\[ M, M :::= x_n P \mid \delta(x_n P, Q) \mid \text{w} M \mid \iota_1 P \mid \iota_2 P \mid \iota_1 P \mid \iota_2 M, \]

with typing rules as follows:

\[ \begin{align*}
    & M_1 : (c_1 \vdash b_1) \quad \ldots \quad M_n : (c_n \vdash b_n) \\
    & (M_1, \ldots, M_n) : (c_1 \rightarrow b_1) \times \ldots \times (c_n \rightarrow b_n) \\
    & P : (c_2 \rightarrow (c_1 \rightarrow p) \times c_0 \Rightarrow c_1) \\
    & x_n P : (c_2 \rightarrow (c_1 \rightarrow p) \times c_0 \Rightarrow p) \\
    & P : (c_2 \times (c_1 \rightarrow d) \times c_0 \Rightarrow c_1) \\
    & Q : (c_3 \times c_2 \times (c_1 \rightarrow d) \times c_0 \Rightarrow (d \Rightarrow b)) \\
    & \delta(x_n P, Q) : (c_3 \times c_2 \times (c_1 \rightarrow d) \times c_0 \Rightarrow b) \\
    & \text{w} M : ((c_1 \rightarrow b_1) \times c_0 \Rightarrow b) \\
    & \iota_1 P : (c_0 \Rightarrow c_1) \\
    & \iota_2 P : (c_0 \Rightarrow c_1 \times c_2) \\
    & \iota_1 P : (c_0 \Rightarrow c_1 \times c_2) \\
    & M : (c_0 \Rightarrow d) \\
    & \iota_2 P : (c_0 \Rightarrow c + d) \\
    & \iota_2 M : (c_0 \Rightarrow c + d). 
\end{align*} \]

The typing rules above involve two kinds of typing judgments.

**Judgments at base type:** Denoted \( M : (c \vdash b) \), this is the main judgment kind, the conclusion of all but the first typing rule. It says that \( M \) is a term of type \( b \) (i.e. either an atomic \( p \) or a disjunction \( d \)) in the typing context \( c \). This context \( c \) takes over the place of the usual context \( \Gamma \) and allows only hypotheses (variables) of type \( c_1 \rightarrow b_1 \) to be used inside the term \( M \).

**Judgments at product type:** Denoted \( P : c \) or \( Q : c \), this kind of judgment is only the conclusion of the first typing rule, whose sole purpose is to make a tuple of base type judgments. However, the judgments at product type are used as hypotheses in the other typing rules, where their role is to allow \( n \) premises to the typing rule. For this usage, they are disguised as the macro-expansions \( c_1 \Rightarrow c_2 \) or \( c \Rightarrow (d \Rightarrow b) \). Implemented by the Coq fixpoints \( \text{explogn} \) and \( \text{explog}1 \), these macro expansions work as follows:

\[ c_0 \Rightarrow (c_1 \rightarrow b_1) \times \ldots \times (c_n \rightarrow b_n) \equiv (c_1 \times c_0 \Rightarrow b_1) \times \ldots \times (c_n \times c_0 \rightarrow b_n) \]

\[ (c_1 + \ldots + c_n) \Rightarrow b \equiv (c_1 \rightarrow b) \times \ldots \times (c_n \rightarrow b) \]

Note that the usage of \( d \Rightarrow b \) in the typing rule for \( \delta \) allows the number of premises contained in \( Q \) to be determined by the size of the sum \( d \).

The typing rules also rely on a implicit variable convention, where a variable \( x_n \) actually denotes the variable whose deBruijn index is \( n \) (we start counting from 0).

**Variables as deBruijn indices:** The variable \( x_n \) in the rules for \( x_n P \) and \( \delta(x_n P, Q) \) represent the hypothesis \( c_1 \rightarrow p \) and \( c_1 \rightarrow d \). For concrete \( c_2, c_3 \), the subscript \( n \) means that the variable represents the \( n \)-th hypothesis of the form \( c \rightarrow b \), counting from left to right and starting from 0, in the context of the term \( P \), or the \( n + 1 \)-st, in the context of the term \( Q \).

We shall motivate our syntax in comparison to the syntax of the lambda calculus from Figure 2 by considering in order all term constructors of the later.

\( x_n \) Since \( e \in \text{ENF} \), either \( e = e \in \text{CNF} \) or \( e = d \in \text{DNF} \).

Variables of type \( d \) only appear as binders in the second branch of \( \delta \), so if we have \( n \)-ary instead of \( n \)-ary the only type variable \( x \) could have will be a \( c \). But, since \( e \) is always of the form \( (c_1 \rightarrow b_1) \times \ldots \times (c_n \rightarrow b_n) \), a variable \( x^n \) could be written as a tuple of \( n \) variables \( x_i \) of types \( c_i \rightarrow b_i \). Moreover, as we want our terms to always be \( \eta \)-expanded, and \( c_1 \rightarrow b_1 \) is an arrow type, we will not have a separate syntactic category of type \( n \) tuple of \( n \) variables. Moreover, but \( n \) will be encoded/merged with either the category of \( n \) variables. In the next new, but \( n \) will be \( \eta \)-expanded, and \( c_1 \rightarrow b_1 \) is an arrow type, we will not have a separate syntactic category of type \( n \) tuple of \( n \) variables. Moreover, but \( n \) will be encoded/merged with either the category of \( n \) variables.

\( M^{e \rightarrow b} N^{e} \) We shall only need this term constructor at type \( b = p \), since if \( b \in \text{DNF} \) the term \( M N \) would not be \( \eta \)-long (we want it to be represented by a \( \delta(M N, \ldots) \)). As we realized during our Coq development, we shall only need the case \( M = x \), as there will be no other syntactic element of type \( c \rightarrow p \) (there will be no projections \( \pi_i \) left, while the \( \delta \) will only be necessary at type \( b \)). In particular, the application \( x\varepsilon \) can be used to represent the old category of variables, where \( \varepsilon \) is the empty tuple of unit type \( 1 \) (the nullary product).

\( \pi_1 M^{(e \rightarrow b) \times c_b \rightarrow c_b} \) If \( M \) is \( \eta \)-expanded, as we want all terms to be, this term would only create a \( \beta \)-redex, and so will not be a part of the new syntax, as we are building a syntax for \( \beta \)-normal and \( \eta \)-long terms.

\( \pi_2 M^{(e \rightarrow b) \times c_b \rightarrow c_b} \) When product types are represented as \( n \)-ary, the same reasoning as for \( \pi_1 \) applies, so \( \pi_2 \) will not be part of the new syntax.

\( \delta(M^{e \rightarrow d}, x^{e_1}, N_{e_1}^{e_2}, x^{e_2}N_{e_2}^{e_2}) \) This constructor is only needed at the type \( (e \rightarrow b) \times \chi \), a consequence of the fact that the \( \eta \)-expanded axioms are only needed for the \( n \)-ary tuple, when the \( \delta \) will be the case of application; this additional restriction was not possible to see upfront, but only once we used Coq to analyze the terms needed for the normalization.

The new constructor \( \delta(x_n P, Q) \) is thus like the old \( \delta(x_n P, \ldots) \), except that \( Q \) regroups in the form of an \( n \)-ary tuple all the
possible branches of the pattern matching (sum types will also be \( n \)-ary, not binary like before).

\[(\lambda x. M)^{c \rightarrow b} \] This terms constructor is already severely restricted (for instance only one variable \( x \) can be abstracted), thanks to the restrictions on the left- and right-hand sides of the function type. But, as we found out during the Coq development, somewhat to our surprise, there is no need for \( \lambda \)-abstraction in our syntax. When reverse-normalizing from our calculus to the standard lambda calculus (see the six examples below), \( \lambda \)'s can be reconstructed thanks to the typing information.

\[(M^{b \rightarrow c})^{N^{a \rightarrow d}} \] This constructor will be maintained, corresponding to the only typing rule with conclusion a judgment of product type, but it will become \( n \)-ary, \( (M_1, \ldots, M_n) \). In particular, we may have the nullary tuple \( \langle \rangle \) of the null product type (i.e. unit type 1).

\[\langle t_1 M^{c \rightarrow d}, (t_2 M^{d \rightarrow e}) \rangle \] These constructors will be maintained, but will be duplicated: the new \( t_{1'}, t_{2'} \) will only be used to construct a binary sum \( c_1 + c_2 \) (this is the base case of sum constructors which must be at least binary by construction), while the \( t_1', t_2' \) will be used to construct sums of the form \( c + d \).

We shall now show a number of examples that our compact term representation manages to represent canonically. We will also show cases when \( \beta \eta \)-equality can not be decided using bringing terms to the compact normal form. For simplicity, all type variables \((a, b, c, d, e, f, g, p, q, r, s, t, j, k, l)\) are assumed to be of atomic type, none of them denoting members of Base, CNF, and DNF, anymore, and for the rest of this subsection.

**Convention 1.** We shall adopt the convention of writing the type \( 1 \rightarrow p \) as \( p \) (1 is the unit type i.e. the nullary product type), writing the application to a nullary pair \( x_n \langle \rangle \) as \( x_n \), and writing a singleton pair \( \langle M \rangle \) as just \( M \). Hence, for instance, an application of some term \( M \) to a singleton pair, containing an application of a term \( N \) to a nullary pair, \( M \langle N \langle \rangle \rangle \), will be written as the more readable \( MN \) corresponding to the usual \( \lambda \)-calculus intuitions.

**Example 1.** This is the first example from the introduction, concerning the relative positions of \( \lambda \)'s, \( \delta \)'s, and applications. The \( \beta \eta \)-equal terms

\[
\begin{align*}
\lambda x. \lambda y. y \delta(x, z, i_1 z, z, i_2 z) = & (8) \\
\lambda x. \lambda y. \delta(x, z, y(g(i_1 z), z, y(i_2 z)) = & (9) \\
\lambda x. \delta(x, z, i_1 z, i_2 z), \lambda z. \lambda y. y(i_2 z) = & (10) \\
\lambda x. y \delta(x, y) = & (11)
\end{align*}
\]

at type

\[
(p + q) \rightarrow ((p + q) \rightarrow r) \rightarrow r
\]

are all normalized to the same canonical representation

\[
\langle x_0 x_2, x_1 x_2 \rangle
\]

at the ENF type

\[
\langle (p \rightarrow r) \times (q \rightarrow r) \times p \rightarrow r \rangle \\
\langle (p \rightarrow r) \times (q \rightarrow r) \times q \rightarrow r \rangle
\]

which can be reverse-normalized back to \( (9) \). However, the point is not that \( (9) \) is somehow better than the other 3 terms, but that a canonical representation should be sought at the ENF type, not the original type! This remark is valid in general, and in particular for the other examples below.

**Example 2.** This is the second example from the introduction (Example 6.2.4.2 from [Balat et al. 2004]). The \( \beta \eta \)-equal terms

\[
\begin{align*}
\lambda y x z u. \delta(x, y, z, u) = & (13) \\
\lambda x y z u. \delta(u, x, y, z) = & (14) \\
\lambda x y z u. \delta(u, x, y, z) = & (15) \\
\lambda x y z u. \delta(u, x, y, z) = & (16)
\end{align*}
\]

at type

\[
\begin{align*}
(a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow (d + e) \rightarrow b
\end{align*}
\]

are all normalized to the compact term

\[
\langle x_3 (x_2 x_1), x_3 (x_2 x_1) \rangle
\]

at the ENF type

\[
\begin{align*}
(d + c \times (c \rightarrow a) \times (a \rightarrow b) \times e \times c \times (c \rightarrow a) \times (a \rightarrow b) \rightarrow b
\end{align*}
\]

which can then be reverse-normalized to \( (14) \), if desired.

A reviewer once remarked that the two previous examples can be handled just by a CPS transformation. While our implementation will be based on continuations, the reason why these examples are handled by our method are not continuations, but rather the fact that all sum types can be eliminated, allowing us to choose a canonical term in the compact representation of the \( \langle \rightarrow, \times \rangle \)-typed lambda calculus.

**Example 3 (Commuting conversions).** The left and right hand sides of the common commuting conversions,

\[
\begin{align*}
\lambda x y z u. \delta(u, v_1, y v_1, z, v_2 v_2) x = & \beta \eta \lambda x y z u. \delta(u, v_1, (y v_1) x, v_2(z v_2) x), \quad (18) \\
\lambda x y z u. \delta(x, x_1 y x_1, x_2, x x_2), w_1 u w_1, w_2 v w_2) = & \beta \eta \lambda x y z u. \delta(x, x_1 y(x_1 w_1, w_1 u w_1, w_2 v w_2), x_2, \delta(x_2, x_1 u w_1, w_1 u w_2, w_2 v w_2)) \quad (19)
\end{align*}
\]

are normalized to the compact terms

\[
\langle x_2 (x_3 x_0), x_1 (x_3, x_0) \rangle, \quad (15)
\]

of types

\[
\langle x_2 (x_3 x_0), x_1 (x_3, x_0) \rangle, \quad (15)
\]

of ENF type

\[
\begin{align*}
(p \times (s \times q \rightarrow r) \times (s \times p \rightarrow r) \times s \rightarrow r) \times \\
(q \times (s \times q \rightarrow r) \times (s \times p \rightarrow r) \times s \rightarrow r)
\end{align*}
\]

and

\[
\langle \delta(x_2 x_4, x_2 x_0, x_1 x_0), \delta(x_2 x_4, x_2 x_0, x_1 x_0) \rangle, \quad (19)
\]

of ENF type

\[
\begin{align*}
(s \rightarrow a) \times (r \rightarrow a) \times (q \rightarrow r + s) \times (p \rightarrow r + s) \times p \rightarrow a) \\
\times ((s \rightarrow a) \times (r \rightarrow a) \times (q \rightarrow r + s) \times (p \rightarrow r + s) \times q \rightarrow a),
\end{align*}
\]

which can be reverse-normalized to the right-hand sides of \( (18) \) and \( (19) \), respectively, if desired.
Example 4 (Eta equations). Both the left- and the right-hand sides of the eta rules (represented as closed terms),

$$\lambda x.x =_\beta \lambda y.yx$$
$$\lambda x.x =_\beta \lambda x.\langle \pi_1 x, \pi x \rangle$$
$$\lambda xy.x \delta(y, x, x \langle x_1, x_1 \rangle)$$
$$\lambda xy.z.\lambda x.z_1 x_1 (z_2, \lambda y.yz_2)$$

of types

of type

are mapped to the same compact term

$$\langle x_1 x_0, x_2 x_0 \rangle$$

of ENF types

and reverse-normalizing these compact terms produces always the right-hand side of the corresponding equation involving lambda terms.

Finally, as we shall see in the following two examples, our conversion to compact form does not guarantee a canonical representation for terms that are equal with respect to the strong forms of $\beta\eta$-equality used to duplicate subterms (Example 5) or change the order of case analysis of subterms (Example 6). Although such term transformations might not be desirable in the setting of real programming languages, for they change the order of evaluation, in a pure effect-free setting like a proof assistant, such transformation would be handy to have.

Example 5. The following $\beta\eta$-equal terms,

$$\lambda xy.z.\delta(uz, ww, xw, wyw)$$

of type

are normalized to two different compact representations:

$$\delta(x_0 x_1, \langle x_2 x_0, x_3 x_0 \rangle)$$

$$\langle \delta(x_0 x_1, \langle x_2 x_0, x_3 x_0 \rangle), x_3 x_0 \rangle$$

Example 6. The following $\beta\eta$-equal terms,

$$\lambda xy.z.\delta(uz, x_1 x_2, x_3 \delta(ww, y_1, y_2 y, y_2 t x_4))$$

of type

are normalized to two different compact representations:

$$\delta(x_2 x_0, \langle x_1 x_3, \langle x_2 x_1, x_3 x_4 \rangle \rangle)$$

$$\langle \delta(x_1 x_0, \langle x_2 x_1, \langle x_1 x_0, x_2 x_3 \rangle \rangle), x_1 x_3 \rangle$$

Comparison to the examples covered by the heuristic of [Balat et al. 2003]. In addition to Example that was borrowed from [Balat et al. 2003], other examples that can be covered from that paper are examples 4.2.1–4.2.4 and Example 4.3.1. In these examples, not only are the input and the output of their TDPE represented uniquely, but also, in the cases when there are two distinct output normal forms according to Balat et al., our normalizer unifies the two normal forms into one, shown below:

$$\langle x_0 \rangle$$

$$\langle x_1 x_0, x_2 x_0 \rangle$$

$$\langle x_1 x_0, x_2 x_0 \rangle$$

The canonical representations could be obtained in these examples, because it was possible to represent the input and output terms in the fragment of the compact calculus which does not include $\delta$'s (although it still involves sum types).

On the other hand, there are also the examples where the input and output are not unified by our procedure\footnote{We shall reproduce the compact representation for these examples in this paper, but they are available for inspection in the Coq formalization accompanying it.} Examples 4.3.2 and 4.3.3 are not handled because we do not permit the order of case analyses (as shown by our Example. Example 6.2.4.2 is not handled because we do not analyze if a subterm has been used twice in a term or not (as shown also by our Example). Examples 4.3.4 and 4.4 are not even executable in our implementation, because we do not have a special treatment of the atomic empty type.

Of course, there is nothing stopping us from applying the program transformations that would allow to handle this kind of...
The forcing structure is an abstract signature (Coq module type), very elegantly, and with getting certain correctness properties for free, using the GADTs from Ocaml’s type system, as shown by the recent work of Danvy, Keller, and Puech [Danvy et al 2015]. Nevertheless, we had chosen to carry out our implementation in Coq, because that allowed us to perform a careful interactive analysis of the necessary normal forms—hence the compact calculus introduced in the first part of this section.

Usual type-directed partial evaluation (TDPE), aka normalization-by-evaluation (NBE), proceeds in two phases. First an evaluator is defined which takes the input term and obtains its semantic representation, and then a reifier is used to map the semantic representation into an output syntactic term. Our TDPE uses an intermediate phase between the two phases, a phase where type isomorphism is respected; intuitively, this has to do with preserving the monotonicity of context free variables: we cannot ‘forget’ a free variable i.e. contexts cannot decrease.

The semantics that we use is defined by a continuation monad over a forcing structure, together with forcing fixpoints that map the type of the input term into a type of the ambient type theory. The forcing structure is an abstract signature (Coq module type), requiring a set $K$ of possible worlds, a preorder relation on worlds, $le$, an interpretation of atomic types, pforsces, and $X$, the return type of the continuation monad.

4.1 A converter for the compact term representation

In the remaining part of this section, we explain the high-level structure of our prototype normalizer of lambda terms into compact terms and vice versa. The full Coq implementation of the normalizer, together with the examples considered above, is given as a companion to this paper. This is only one possible implementation, using continuations, but all the previous material of this paper was written as generically as possible, so that it is useful if other implementation techniques are attempted in the future, such as rewriting based on evaluation contexts (i.e. the first-order reification of continuations), or abstract machines.

In a nutshell, our implementation is a type-directed partial evaluator, written in continuation-passing style, with an intermediate phase between the evaluation and reification phases, that allows to map a ‘semantic’ representation of a term from a type to its ENF type, and vice versa. Such partial evaluators can be implemented very elegantly, and with getting certain correctness properties for free, using the GADTs from Ocaml’s type system, as shown by the recent work of Danvy, Keller, and Puech [Danvy et al 2015]. Nevertheless, we had chosen to carry out our implementation in Coq, because that allowed us to perform a careful interactive analysis of the necessary normal forms—hence the compact calculus introduced in the first part of this section.

Usual type-directed partial evaluation (TDPE), aka normalization-by-evaluation (NBE), proceeds in two phases. First an evaluator is defined which takes the input term and obtains its semantic representation, and then a reifier is used to map the semantic representation into an output syntactic term. Our TDPE uses an intermediate phase between the two phases, a phase where type isomorphisms are applied to the semantic domain so that the narrowing down of a class of equal terms, described in Section 2, is performed on the semantic annotation of a term.

The semantics that we use is defined by a continuation monad over a forcing structure, together with forcing fixpoints that map the type of the input term into a type of the ambient type theory. The forcing structure is an abstract signature (Coq module type), requiring a set $K$ of possible worlds, a preorder relation on worlds, $le$, an interpretation of atomic types, pforsces, and $X$, the return type of the continuation monad.

The continuation monad is polymorphic and instantiable by a forcing fixpoint $f$ and a world $w$. It ensures that the preorder relation is respected; intuitively, this has to do with preserving the monotonicity of context free variables: we cannot ‘forget’ a free variable i.e. contexts cannot decrease.

The forcing structure is an abstract signature (Coq module type), very elegantly, and with getting certain correctness properties for free, using the GADTs from Ocaml’s type system, as shown by the recent work of Danvy, Keller, and Puech [Danvy et al 2015]. Nevertheless, we had chosen to carry out our implementation in Coq, because that allowed us to perform a careful interactive analysis of the necessary normal forms—hence the compact calculus introduced in the first part of this section.

Usual type-directed partial evaluation (TDPE), aka normalization-by-evaluation (NBE), proceeds in two phases. First an evaluator is defined which takes the input term and obtains its semantic representation, and then a reifier is used to map the semantic representation into an output syntactic term. Our TDPE uses an intermediate phase between the two phases, a phase where type isomorphisms are applied to the semantic domain so that the narrowing down of a class of equal terms, described in Section 2, is performed on the semantic annotation of a term.

The semantics that we use is defined by a continuation monad over a forcing structure, together with forcing fixpoints that map the type of the input term into a type of the ambient type theory. The forcing structure is an abstract signature (Coq module type), requiring a set $K$ of possible worlds, a preorder relation on worlds, $le$, an interpretation of atomic types, pforsces, and $X$, the return type of the continuation monad.

Given these definitions, we can write an evaluator for compact terms, actually two simultaneously defined evaluators evalc and evalb, proceeding by induction on the input term.

An evaluator for usual lambda terms can also be defined, by induction on the input term. A helper function lforces analogous to the list map function for sforsces is necessary.
The novelty of our implementation (besides isolating the compact term calculus itself), in comparison to previous type-directed partial evaluators for the lambda calculus with sums, consists in showing that one can go back and forth between the semantic annotation at a type \( F \) and the semantic annotation of the normal form \( \text{enf}(F) \). The proof of this statement needs a number of auxiliary lemmas that we do not mention in the paper. We actually prove two statements simultaneously, \( f2f \) and \( f2f' \), declared as follows.

\[\text{Theorem } f2f : \quad \forall F, \forall w, \text{Cont sforces w F} \rightarrow \text{Cont eforces w (enf F)}\]
\[\text{with } f2f' : \quad \forall F, \forall w, \text{Cont eforces w (enf F)} \rightarrow \text{Cont sforces w F}.\]

As one can see from their type signatures, \( f2f \) and \( f2f' \) provide a link between the semantics of the standard lambda calculus for sums (ND) and the semantics of our compact calculus (HS/HSb).

We move forward to describing the reification phase. In this phase, two instantiations of a forcing structure are needed. Unlike the evaluators, which can work over an abstract forcing structure, the reifiers need concrete instantiations built from the syntax of the term calculus in order to produce syntactic normal forms.

The first instantiation is a forcing structure for the standard lambda calculus with sums. The set of worlds is the set of contexts (lists of types), the preorder on worlds is defined as the prefix relation on contexts, the forcing of an atomic type \( p \) is the set of terms of type \( p \) in the context \( w \), and the answer type of the continuation monad is the set of terms of type \( F \) in the context \( w \). One could be more precise, and instantiate the answer type by the set of normal/neutral terms, like it has been done in most other implementations of TDPE, and in our own prior works, but for the sake of simplicity, we do not make that distinction in this paper.

\[\text{Inductive } le_\_ : \text{list Formula} \rightarrow \text{list Formula} \rightarrow \text{Set} :=\]
\[\text{with } f2f' : \quad \forall F, \forall w, \text{Cont eforces w (enf F)} \rightarrow \text{Cont sforces w F}.\]

The second instantiation is a forcing structure for our calculus of compact terms. The set of worlds is the same as the set of CNFs, because our context are simply CNFs, the preorder is the prefix relation on CNFs, the forcing of atomic types are terms of atomic types, and the answer type of the continuation monad is the set of terms at base type.

\[\text{Inductive } le_\_ : \text{list Formula} \rightarrow \text{list Formula} \rightarrow \text{Set} :=\]
\[\text{with } f2f' : \quad \forall F, \forall w, \text{Cont eforces w (enf F)} \rightarrow \text{Cont sforces w F}.\]

Using the instantiated forcing structures, we can provide reification functions for terms of the lambda calculus.

\[\text{Theorem } sreify : \quad \forall F w, \text{Cont sforces w F} \rightarrow \text{ND w F}\]
\[\text{with } srereflect : \quad \forall F w, \text{ND w F} \rightarrow \text{Cont sforces w F}.\]

The reifier for atomic types, \( pforces \), is not listed above, because it is simply the ‘run’ operation on the continuation monad. As usually in TDPE, every reification function required its own simultaneously defined reflection function.

Finally, one can combine the reifiers, the evaluators, and the functions \( f2f \) and \( f2f' \), in order to obtain both a normalizing converter of lambda terms into compact terms (called \( \text{ebn} \) in the Coq implementation), and a converter of compact terms into lambda terms (called \( \text{ebn} \) in the Coq implementation). One can, if one desires, also define only a partial evaluator of lambda terms and only a partial evaluator of compact terms.

5. Conclusion

Summary of our results. We have brought into relation two distinct fundamental problems of the lambda calculi underlying modern functional programming languages, one concerning identity of types, and the other concerning identity of terms, and we have shown how improved understanding of the first problem can lead to improved understanding of the second problem.

We started by presenting a normal form of types, the exp-log normal form, that is a systematic ordering of the high-school identities allowing for a type to be mapped to normal form. This can be used as a simple heuristic for deciding type isomorphism, a first such result for the type language \( \rightarrow, +, \times \). We believe that the link established to analysis and abstract algebra (the exp-log decomposition produces a pair of homomorphisms between the additive and the multiplicative group in an exponential field) may also be beneficial to programming languages theory in the future.

The typing restrictions imposed to lambda terms in exp-log normal form allowed us to decompose the standard axioms for \( \beta_0 \) into a proper and simpler subset of themselves, \( \beta_0 \). As far as we are aware, this simpler axiomatization has not been isolated before. 

\[\text{Inductive } le_\_ : \text{list Formula} \rightarrow \text{list Formula} \rightarrow \text{Set} :=\]
\[\text{with } f2f' : \quad \forall F, \forall w, \text{Cont eforces w (enf F)} \rightarrow \text{Cont sforces w F}.\]
Even more pleasingly, the new axiomatization disentangles the old one, in the sense that left-hand sides and right-hand sides of the equality axioms can no longer overlap.

Finally, we ended by giving a compact calculus of terms that can be used as a more canonical alternative to the lambda calculus when modeling the core of functional programming languages: the new syntax does not allow for the \( \eta \)-axioms of Figure 2—even to be stated, with the exception of \( \eta_m \) that is still present, albeit with a restricted type. As our method exploits type information, it is orthogonal to the existing approaches that rely on term analysis (discussed below), and hence could be used in addition to them; we hope that it may one day help with addressing the part of \( \eta \)-equality that is still beyond decision procedures. We also implemented and described a prototype converter from/to standard lambda terms.

In the future, we would like to derive declarative rules to describe more explicitly the extent of the fragment of \( =_{\beta\eta} \) decided by our heuristic, although implicitly that fragment is determined by the reduction to ENF congruence classes explained in Section 3. It should be noted that in this respect our heuristic is no less explicitly described, than the only other published one (Balat et al. 2004) (reviewed below).

**Related work** Dougherty and Subrahmanyan (Dougherty and Subrahmanyan 1993) show that the equational theory of terms (morphisms) for almost bi-Cartesian closed categories is complete with respect to the set theoretic semantics. This presents a generalization of Friedman’s completeness theorem for simply typed lambda calculus without sums (Cartesian closed categories) (Friedman 1975). Ghani (Ghani 1998) proves \( \beta\eta \)-equality of terms of the lambda calculus with sum types to be decidable, first proceeding by rewriting and eta-expansion, and then checking equality up to commuting conversions by interpreting terms as finite sets of quasi-normal forms; no canonical normal forms are obtained.

When sums are absent, the existence of a confluent and strongly normalizing rewrite system proves the existence of canonical normal forms, and then decidability is a simple check of syntactic identity of canonical forms. Nevertheless, even in the context when sums are absent, one may be interested in getting term representations that are canonical modulo type isomorphism, as in the recent works of Díaz-Caro, Dowek, and Martínez López (Díaz-Caro and Dowek 2015; Díaz-Caro and Martínez López 2016). Altenkirch, Dybjer, Hofmann, and Scott (Altenkirch et al. 2001) give another proof of decidability of \( \beta\gamma \)-equality for the lambda calculus with sums by carrying out a normalization-by-evaluation argument in category theory. They provide a canonical interpretation of the syntax in the category of sheaves for the Grothendieck topology over the category of constrained environments, and they claim that one can obtain an algorithm for a decision procedure by virtue of the whole development being formalizable in extensional Martin-Löf type theory.

In the absence of \( \eta_m \) (Dougherty 1993), or for the restriction of \( \eta_m \) to be a variable (Di Cosmo and Kesner 1993), a confluent and strongly normalizing rewrite system exists, hence canonicity of normal forms for such systems follows.

In (Balat et al. 2004), Balat, Di Cosmo, and Fiore, present a notion of normal form which is a syntactic counterpart to the notion of normal forms in sheaves of Altenkirch, Dybjer, Hofmann, and Scott. However, the forms are not canonical, as there may be two different syntactic normal forms corresponding to a single semantic one. They also say they believe (without further analysis or proof) that one can get canonical normal forms if one considers an ordering of nested \( \delta \)-expressions.

The normal forms of Balat et al. are sophisticated and determining if something is a normal form relies on comparing sub-terms up to a congruence relation \( \approx \) on terms; essentially, this congruence allows to identify terms such as the ones of our Example 5 and Example 6. For determining if something is a normal form, in addition to the standard separation of neutral vs normal terms, one uses three additional criteria: (A) \( \delta \)-expressions that appear under a lambda abstraction must only case-analyze terms involving the abstracted variable; (B) no two terms which are equal modulo \( \approx \) can be case analyzed twice; in particular, no term can be case analyzed twice; (C) no case analysis can have the two branches which are equal modulo \( \approx \). To enforce condition A, particular powerful control operators, set/cupto, are needed in the implementation, requiring a patch of the ocaml toplevel. Using our compact terms instead of lambda terms should help get rid of condition A (hence set/cupto), since as we showed keeping a constructor for \( \lambda \)’s in the representation of normal forms is not necessary. On the other hand, we could profit from implementing checks such as B and C in our implementation; however, our goal was to see how far we can get in a purely type-directed way without doing any program analysis.

A final small remark about this line of works: in (Balat 2009), Balat used the word “canonical” to name his normal forms, but this does not preserve the usual meaning of that word, as showed in the previous article (Balat et al. 2004).

Lindley (Lindley 2007) presents another proof of decidability of \( \beta\gamma \)-equality for the lambda calculus with sums, based on an original axiomatization of the \( \eta \)-axiom into four axioms involving evaluation contexts (the proof of this decomposition, Proposition 1, is unfortunately only sketched); the proof of decidability uses rewriting modulo the congruence relation \( \approx \) of Balat et al.

Scherer (Scherer 2015) reinterprets Lindley’s rewriting approach to decidability in the setting of the structural proof theory of maximal multi-focusing, where he brings it in relation to the technique of preemptive rewriting (Chaudhuri et al. 2008). Scherer seems to derive canonicity of his normal forms for natural deduction from Lindley’s results, although the later does not seem to show canonical forms are a result of his rewriting decidability result.

The idea to apply type isomorphism in order to capture equality of terms has been used before (Ahmad et al. 2010), but only implicitly. Namely, in the focusing approach to sequent calculi (Liang and Miller 2007), one gets a more canonical representation of terms (proofs) by grouping all so called asynchronous proof rules into blocks called asynchronous phases. However, while all asynchronous proof rules are special kinds of type isomorphisms, not all possible type isomorphisms are accounted for by the asynchronous blocks: sequent calculi apply asynchronous proof rules superficially, by looking at the top-most connectives, but normalizing sequents (formulas) to their exp-log normal form applies proof rules deeply inside the proof tree. Our approach can thus also been seen as moving focusing proof systems into the direction of so called deep inference systems.

**References**


