

Exceptional NbE for Sums

Freirc Barral

Department of Computer Science,
Ludwig-Maximilians-University Munich,
Oettingenstraße 67, 80 538 München, Germany
barral@tcs.ifl.lmu.de

Abstract

We devise an algorithm using normalization by evaluation (NbE) for deciding equality between terms in a λ -calculus equipped with strong sums. While proofs of correctness and completeness are still work in progress, the originality of this algorithm is its use of exceptions, which yields a particularly simple solution to this intricate problem.

1. Introduction

Although the extension of the simply typed λ -calculus with sum types is seemingly simple (after all we are only adding the possibility to do case analysis into the system), the study of its theory is extremely complex (for a general introduction of the difficulties encountered, see [9]). These difficulties depend on which kind of rules one wants to add besides β -reduction. For the extension by permutative conversion, the system is proved strongly normalizing by a CPS-translation in [8], by a variant of the reducibility candidate method in [14], and by an original syntax directed characterization of the strong normalizable terms in [13]. For the extension to a full extensional system, the permutation of independent case expressions prevents the design of a reasonable system with the confluence and strong normalization property. Nevertheless, Neil Ghani proposes a proof of decidability entirely based on rewriting theory [11].

Another way to decide equality in typed λ -calculus, particularly well-suited to incorporate extensional rules, is to use normalization by evaluation. The first algorithm of normalization by evaluation for a system with sum types is given by Olivier Danvy in [7], using continuations and control operators. Andrzej Filinski presents an algorithm in [10], as well based on continuations and control operators, but in a monadic setting, which is of particular interest to us.

In [1], Thorsten Altenkirch, Peter Dybjer, Martin Hofmann and Philip Scott prove constructively the existence of an NbE algorithm, able to decide the whole extensional theory of a simply typed lambda extended with sum types, and hence a direct program extraction from a formalization of the proof should provide the algorithm. An algorithm for an extension by a Boolean type together with the proof of its correctness and completeness is given in [2]. The challenge to give a practical NbE algorithm for a system with general sum types, and for a conversion stronger than merely beta was first taken on by Vincent Balat and Olivier Danvy in [5]. This approach using continuations and control operators is then further pursued by Vincent Balat in [3] and his coauthors Roberto Di Cosmo, and Marcelo Fiore in [4]. In these latter works, the algorithm is optimized and the set of normal forms resulting from the algorithm is further constrained, and this allows to design an algorithm for the whole extensional conversion of the calculus.

Our work is based both on the approach developed by Altenkirch and his coauthors and Balat and his coauthors, we will present an algorithm based on normalization by evaluation for the normalization of terms in a system with sum types and extensional conversions. However, our algorithm is based on the notion of exceptions, which we consider as a much simpler framework than continuations (exceptions can be seen as trivial continuations).

Moreover we do not use any control operators or imperative effects such as assignment. Although for the commodity of the exposition, we will use a pseudo programming language where exceptions are a built-in feature, the algorithm is presented in a pure functional style, i.e., in a monadic setting, in the work of the author [6], in the continuation of Filinski [10], but for a stronger conversion relation.

2. System

In this first section we present the system Λ_+ , which is an extension of the simply typed λ -calculus with sum types.

A sum type of two types ρ and σ will be interpreted as a disjoint union of the interpretations of ρ and σ .

Types and terms

Given a base type o , the set Ty of Types is defined by the following inductive definition:

$$\text{Ty} \ni \rho, \sigma ::= o \mid \rho \rightarrow \sigma \mid \rho + \sigma$$

Given a countable infinite set of terms variables Var , the set of terms Tm of the simply typed λ -calculus with sum type is defined inductively by:

$$\text{Tm} \ni r, s, t ::= x \mid \lambda x^\rho. r \mid rs \mid \text{in}_i r \mid \delta(r, x^\rho. s, y^\sigma. t)$$

where $x \in \text{Var}$

Notation. We have chosen to express our system in Church style, which means that types of bound variables are explicitly given at the binding symbol. The dot after a variable in a branch of a δ term binds the variable in the branch, hence this variable is written with its type. However, for readability, we will allow us to omit these types when they are clear from the context.

Because terms containing case symbols δ tends to become large, we will use sometimes an alternative vertical notation $\delta\left(r, \begin{smallmatrix} x.s \\ y.t \end{smallmatrix}\right)$ for $\delta(r, x.s, y.t)$.

The set of free variable of a term r will be noted $\text{FV}(r)$.

Typing

The typing relation is a ternary relation $\Gamma \vdash r : \rho$ between a context Γ , a term r and a type ρ and is defined inductively as follows:

$$\frac{(x : \rho) \in \Gamma}{\Gamma \vdash x : \rho} (\text{VAR}) \quad \frac{\Gamma, x : \rho \vdash r : \sigma}{\Gamma \vdash \lambda x^\rho. r : \rho \rightarrow \sigma} (\rightarrow\text{I})$$

$$\frac{\Gamma \vdash r : \rho \rightarrow \sigma \quad \Gamma \vdash s : \rho}{\Gamma \vdash rs : \sigma} (\rightarrow\text{E}) \quad \frac{\Gamma \vdash r : \rho_i}{\Gamma \vdash \text{in}_i r : \rho_0 + \rho_1} (+\text{I})$$

$$\frac{\Gamma \vdash r : \rho_0 + \rho_1 \quad \Gamma, x_0 : \rho_0 \vdash s_0 : \sigma \quad \Gamma, x_1 : \rho_1 \vdash s_1 : \sigma}{\Gamma \vdash \delta(r, x_0^{\rho_0}.s_0, x_1^{\rho_1}.s_1) : \sigma} (+\text{E})$$

In the terminology of natural deduction, the typing rules are separated into two classes: those where a type constructor (here \rightarrow or $+$) appears in a premise are called elimination rules, whereas the ones where the type constructor appears in the conclusion are called introduction rules. The premiss of an elimination rule carrying the eliminated type constructor is called a major premiss, and the others are called minor premisses.

Conversions

The theory of the Λ_+ -calculus is defined with a list of axioms. The actual conversion is obtained from these axioms as the smallest congruence relation on the term structure containing these axioms, i.e., a relation which is an equivalence relation and closed under term formation rules.

The axioms for β -conversions are defined as usual.

$$(\lambda x. r)s =_\beta r\{s/x\} \quad (\beta_{\rightarrow})$$

$$\delta(\text{in}_i r, x_0.s_0, x_1.s_1) =_\beta s_i\{r/x_i\} \quad (\beta_+)$$

As mentioned in the introduction, the theory augmented by permutative conversion has already been studied by several authors. This theory allows to identify more terms than the β -conversion alone and is useful in a natural deduction formulation of logic with disjunction because they allow to recover a variant of the subformula property (see [16] or [12] for details). The axioms of the *permutative conversion* are given by:

$$\delta\left(\delta\left(r, \begin{smallmatrix} x_0.s_0 \\ x_1.s_1 \end{smallmatrix}\right), y_0.t_0, y_1.t_1\right) =_\pi \delta\left(r, \begin{smallmatrix} x_0.\delta(s_0, y_0.t_0, y_1.t_1) \\ x_1.\delta(s_1, y_0.t_0, y_1.t_1) \end{smallmatrix}\right) \quad (\pi_1)$$

$$\delta(r, x_0.s_0, x_1.s_1)t =_\pi \delta(r, x_0.s_0t, x_1.s_1t) \quad (\pi_2)$$

where in the axiom π_2 , neither x_0 nor x_1 occurs free in t .

Given a term r , the π -conversion pulls out to the exterior of the term a test term s of a δ -subterm $\delta(s, x_0.s_0, x_1.s_1)$ only if it occurs in certain position of the term r (i.e., in the major premiss). This occurrence conditions of the δ -subterm can be relaxed, leading to the generalisation π^e below of the π conversion.

The axiom of the *extensional permutative conversion* is given by:

$$r\{\delta(s, y_0.t_0, y_1.t_1)/x\} =_{\pi^e} \delta(s, y_0.r\{t_0/x\}, y_1.r\{t_1/x\}) \quad (\pi)$$

where neither y_0 nor y_1 occurs free in r .

Remark. We use a capture avoiding substitution in the definition of the π^e -conversion for sum type above so that the free variables of s can not be bound in r .

The axioms of η -conversion, the axioms η_{\rightarrow} and η_+ are given by

$$r =_{\eta_{\rightarrow}} \lambda x. rx \quad (\eta_{\rightarrow})$$

$$r =_{\eta_+} \delta(r, x_0.\text{in}_0 x_0, x_1.\text{in}_1 x_1) \quad (\eta_+)$$

where in the axiom η_{\rightarrow} , x does not occur free in r . The η_+ -conversion is the analogue of the η_{\rightarrow} -conversion for sum types. It takes an arbitrary terms of sum types and converts

it to a term involving the term constructor for sum type, i.e., injections.

The existence of a strongly normalizing and confluent rewriting system generating this theory is in fact very unlikely (the only work based on rewriting theory to decide this theory [11] does not devise a strongly normalizing system).

3. Extended Conversions and normal forms

In the definition of the π^e -conversion above, because of the substitution, we have to consider the term at an arbitrary depth. We give here a local decomposition of this conversion, which will help us to explain difficulties inherent to the extensional calculus with sum types and to justify the normal form of the term as computed by the *NbE* algorithm. The extensional permutative conversion can be defined by the following axioms

$$\begin{aligned}
\delta\left(\delta\left(r, \begin{array}{l} x_0.s_0 \\ x_1.s_1 \end{array}, y_0.t_0\right), y_1.t_1\right) &=_{\pi_1^e} \delta\left(r, \begin{array}{l} x_0.\delta(s_0, y_0.t_0, y_1.t_1) \\ x_1.\delta(s_1, y_0.t_0, y_1.t_1) \end{array}\right) \\
&\quad \text{where } x_0, x_1 \notin \text{FV}(t_0, t_1) \\
\delta(r, x_0.s_0, x_1.s_1)t &=_{\pi_2^e} \delta(r, x_0.s_0t, x_1.s_1t) \\
&\quad \text{where } (x_0, x_1 \notin \text{FV}(t)) \\
t\delta(r, x_0.s_0, x_1.s_1) &=_{\pi_3^e} \delta(r, x_0.t\ s_0, x_1.t\ s_1) \\
\text{in}_i\delta(r, x_0.s_0, x_1.s_1) &=_{\pi_4^e} \delta(r, x_0.\text{in}_i\ s_0, x_1.\text{in}_i\ s_1) \\
\lambda x.\delta(r, x_0.s_0, x_1.s_1) &=_{\pi_5^e} \delta(r, x_0.\lambda x.s_0, x_1.\lambda x.s_1) \\
&\quad \text{where } x \notin \text{FV}(r) \\
\delta\left(r, \begin{array}{l} x_0.\delta(s, y_0.t_0, y_1.t_1) \\ x_1.u \end{array}\right) &=_{\pi_6^e} \delta\left(s, \begin{array}{l} y_0.\delta(r, x_0.t_0, x_1.u) \\ y_1.\delta(r, x_0.t_1, x_1.u) \end{array}\right) \\
&\quad \text{where } y_0, y_1 \notin \text{FV}(u), x_0 \notin \text{FV}(s) \\
\delta\left(r, \begin{array}{l} x_0.u \\ x_1.\delta(s, y_0.t_0, y_1.t_1) \end{array}\right) &=_{\pi_7^e} \delta\left(s, \begin{array}{l} y_0.\delta(r, x_0.u, x_1.t_0) \\ y_1.\delta(r, x_0.u, x_1.t_1) \end{array}\right) \\
&\quad \text{where } y_0, y_1 \notin \text{FV}(u), x_1 \notin \text{FV}(t) \\
\delta\left(r, \begin{array}{l} x_0.\delta(r, x_0.s_0, x_1.s_1) \\ x_1.t \end{array}\right) &=_{\pi_8^e} \delta(r, x_0.s_0, x_1.t) \\
\delta\left(r, \begin{array}{l} x_0.s \\ x_1.\delta(r, x_0.t_0, x_1.t_1) \end{array}\right) &=_{\pi_9^e} \delta(r, x_0.s, x_1.t_1) \\
&\quad r =_{\text{is}} \delta(s, x_0.r, x_1.r) \\
&\quad \text{where } x_i \notin \text{FV}(r)
\end{aligned}$$

The seven first conversions (π_1^e to π_7^e) above are just instantiations of the general π^e -conversion. The structural closure of these rules generates the general π^e -conversion where substitution is restricted to one occurrence of a variable. The conversion π_8^e and π_9^e handles the case of multiple occurrences whereas the conversion is handles the case of zero occurrence.

Whereas the *NbE* algorithm is not based on rewriting theory, the normal forms it produces can be understood, up to a certain stage, as the irreducible terms of a rewriting system obtained by orienting the conversions above.

The normal forms that we will eventually present are essentially the same as in the works of Balat [3] and Altenkirch, Dybjer, Hofmann and Scott [1].

First we want our normal forms to be a restriction of the existing normal forms for the traditional permutative conversion (given as π in the system section or π_1^e and π_2^e in the alternative definition of π^e -conversion above). The set of normal forms NF_{Λ^+} for permutative conversions π and β -conversions is defined inductively together with a set of neutral terms Ne_{Λ^+} as follows:

$$\begin{aligned}
\text{Ne}_{\Lambda^+} \ni n &::= x \mid nN \\
\text{NF}_{\Lambda^+} \ni N &::= n \mid \lambda x.N \mid \text{in}_i N \mid \delta(n, x_1.N_1, x_0.N_0)
\end{aligned}$$

The rule π_3^e , read from left to right, transforms an application term $t\delta(r, x_0.s_0, x_1.s_1)$ in a δ -term $\delta(r, x_0.t\ s_0, x_1.t\ s_1)$. This is already the effect of the traditional permutative conversion π , so that we keep this direction. Similarly, the rule π_4^e move injections $\text{in}_i\delta(r, x_0.s_0, x_1.s_1)$ inside a δ -term $\delta(r, x_0.\text{in}_i\ s_0, x_1.\text{in}_i\ s_1)$. The normal forms NF need once more to be split and are defined simultaneously with neutral forms Ne and PNF .

$$\begin{aligned}
\text{Ne}_{\Lambda^+} \ni n &::= x \mid nP \\
\text{PNF}_{\Lambda^+} \ni P &::= n \mid \lambda x.N \mid \text{in}_i P \\
\text{NF}_{\Lambda^+} \ni N, N_0, N_1 &::= P \mid \delta(n, x_1.N_1, x_0.N_0)
\end{aligned}$$

The conversion π_5^e says that a λ -abstraction can be moved inside a case term. We choose to move the binders λ as deep as possible in order to introduce variables just before their uses. For that, we need to define a condition which express that a λ is already at the deepest possible position.

For example, for the term $\lambda z.\delta(r, x.s, y.t)$, this condition should state that the variable z must occur in r , otherwise λz could be moved in front of s and t . However, to check whether the variable z occurs in r is not enough, for if the term s is itself a δ -term, say $s = \delta(r', x'.s', y'.t')$, and neither z nor x occurs in r' then we could move λz in a more internal position as follows:

1. first move the term r' to the outside by converting $\lambda z.\delta(r, x.\delta(r', x'.s', y'.t'), y.t)$ into $\lambda z.\delta(r', x'.\delta(r, x.s', y.t), y'.\delta(r, y'.t', y.t))$,
2. and now we can move λz inside by converting $\lambda z.\delta(r', x'.\delta(r, x.s', y.t), y'.\delta(r, y'.t', y.t))$ into $\delta(r', x'.\lambda z.\delta(r, x.s', y.t), y'.\lambda z.\delta(r, y'.t', y.t))$

Therefore we define the deepest λ condition $\text{DL}(\mathbb{X}, r)$, for a set of variables \mathbb{X} and a term r , this condition is true if r is a variable or a λ -abstraction and if r is a case term $\delta(n, x_0.N_0, x_1.N_1)$, it is defined as follows:

$$\begin{aligned}
\text{DL}(\mathbb{X}, \delta(n, x_0.N_0, x_1.N_1)) &::= \mathbb{X} \cap \text{FV}(n) \neq \emptyset \wedge \\
&\quad \text{DL}(\mathbb{X} \cup x_0, N_0) \wedge \text{DL}(\mathbb{X} \cup x_1, N_1)
\end{aligned}$$

Conversion η and prenormal forms

The rule of η -conversion, oriented as an expansion (i.e., from left to right in the definition above), expands a term r into an abstraction or a δ -term according to whether the type of r is an arrow type or a sum type.

Considered together with the β -conversion oriented in a reduction from left to right, one has to impose restrictions on the η_{\rightarrow} -expansion to prevent infinite sequence of reductions. Namely an η_{\rightarrow} -expansion should not expand a λ -abstraction nor the major premiss r of an application $r.s$. Similarly an η_{+} -expansion should not expand an injection term $\text{in}_i r$ nor the major premiss r of a δ term $\delta(r, x.s, y.t)$.

To obtain normal forms for the η -conversion, it suffices to restrict the formation of pure normal forms for neutral terms to those of ground type. We will need to further restrict these normal forms in the next section and call prenormal forms the normal forms obtained at this stage.

Definition 1 (prenormal forms). *We define by simultaneous induction three relations $-\vdash_{\text{Ne}} - : \rho$, $-\vdash_{\text{PNF}} - : \rho$, and $-\vdash_{\text{NF}} - : \rho$ between context, term and type,*

$$\begin{array}{c} \frac{}{\Gamma \vdash_{\text{Ne}} x : \rho} \quad \frac{\Gamma \vdash_{\text{Ne}} n : \rho \rightarrow \sigma \quad \Gamma \vdash_{\text{PNF}} P : \rho}{\Gamma \vdash_{\text{Ne}} nP : \sigma} \\ \\ \frac{\Gamma \vdash_{\text{Ne}} n : \rho}{\Gamma \vdash_{\text{PNF}} n : \rho} \quad \frac{\Gamma, x : \rho \vdash_{\text{NF}} N : \sigma}{\Gamma \vdash_{\text{PNF}} \lambda x.N : \rho \rightarrow \sigma} \\ \\ \frac{\Gamma \vdash_{\text{PNF}} P : \rho_i}{\Gamma \vdash_{\text{PNF}} \text{in}_i P : \rho_0 + \rho_1} \quad \frac{\Gamma \vdash_{\text{PNF}} P : \rho}{\Gamma \vdash_{\text{NF}} P : \rho} \\ \\ \frac{\Gamma \vdash_{\text{Ne}} n : \rho_0 + \rho_1 \quad \Gamma, x : \rho_i \vdash_{\text{NF}} N_i : \sigma}{\Gamma \vdash_{\text{NF}} \delta(n, x_0.N_0, x_1.N_1) : \sigma} \end{array}$$

in the rule with conclusion $\Gamma \vdash_{\text{PNF}} \lambda x.N : \rho \rightarrow \sigma$ we impose the condition

$$\text{DL}(\{x\}, N) \quad (\text{deepest } \lambda)$$

which is true if N is not a δ term and otherwise defined by:

$$\begin{aligned} \text{DL}(\mathbb{X}, \delta(n, x_0.N_0, x_1.N_1)) &::= \mathbb{X} \cap \text{FV}(n) \neq \emptyset \wedge \\ &\text{DL}(\mathbb{X} \cup x_0, N_0) \wedge \text{DL}(\mathbb{X} \cup x_1, N_1) \end{aligned}$$

The condition (deepest λ) ensures that the δ -operators are introduced as soon as possible, i.e., as soon as variables occurring in the term to be tested are available, which means just after a binding symbols or at the top-level.

Notation. We will write $\text{Ne}_{\Gamma}^{\rho}$ (resp. $\text{NF}_{\Gamma}^{\rho}$ and $\text{PNF}_{\Gamma}^{\rho}$) for the set of terms r such that $\Gamma \vdash_{\text{Ne}} r : \rho$ (resp. $\Gamma \vdash_{\text{NF}} r : \rho$ and $\Gamma \vdash_{\text{PNF}} r : \rho$) and Ne^{ρ} (resp. NF^{ρ} and PNF^{ρ}) for the set of terms r such that there exists a context Γ and $r \in \text{Ne}_{\Gamma}^{\rho}$ (resp. $r \in \text{NF}_{\Gamma}^{\rho}$ and $r \in \text{PNF}_{\Gamma}^{\rho}$).

Circular Conversions and Immediate Simplification

To design a decision algorithm for the whole conversion relation, one has to face a first problem, namely the inherent circularity of the π^e -conversion.

As one can see in the following example, this circularity problem cannot be solved by the orientation of the rewrite relation alone.

Example 1. *In the example below the term on the right hand side is obtained by pulling out the two occurrences of the subterm r' thanks to the conversion π_6^e and π_7^e , then the term is simplified by using π_8^e or π_9^e , and by reducing the generated β -redex:*

$$\delta \left(r, \frac{x.\delta(r', x'.s', y'.t')}{y.\delta(r', x'.s'', y'.t'')} \right) = \delta \left(r', \frac{x'.\delta(r, x.s', y.s'')}{y'.\delta(r, x.t', y.t'')} \right)$$

with $x', y' \notin \text{FV}(r)$ and $x, y \notin \text{FV}(r')$.

The two terms of the example above have exactly the same structure, and although it may be possible to break this circularity, e.g., by introducing an order on terms (for example induced by the variables they contain), it seems artificial to select one of these terms as more normal than the other. In terms of programming if we have to do two tests which are independent from each other, there is no reason to choose to do a particular one first.

Hence, the normal forms can no longer be unique, their equality is no longer syntactic but has to be tested with respect to this conversions (i.e., π_6^e to π_9^e).

Notation. *Due to this circularity, we will call the conversions π_6^e to π_9^e circular permutative conversions and write π_c^e for their union as relations.*

$$r =_{\pi_c^e} s ::= r =_{\pi_6^e} s \vee r =_{\pi_7^e} s \vee r =_{\pi_8^e} s \vee r =_{\pi_9^e} s$$

A second problem is that the NbE algorithm tends to produce terms which are in expanded form, i.e., the π^e -conversion is oriented from left to right. In particular this means that if x does not actually occur in r , the term $r\{\delta(s, x_0.t_0, x_1.t_1)/x\}$ is converted into $\delta(s, x_0.r, x_1.r)$ (is-conversion).

Seen as a reduction from left to right this would lead immediately to an infinite loop. Besides, it does not make any sense to introduce superfluous test term as s above. Hence the normal forms will have to be irreducible terms for the inverse reduction (oriented from right to left) known from Prawitz [15] as *immediate simplification reduction*.

$$\delta(s, x_0.r, x_1.r) \longrightarrow_{\text{is}} r \quad (x_0, x_1 \notin \text{FV}(r))$$

We will deal with these conversions in more detail after having defined a first version of the normalization algorithm for the prenormal forms defined above.

4 Informal Description

First, we need an *interpretation* $\llbracket \rho \rrbracket$ of a type ρ . We take the standard set theoretical interpretation where a sum type is interpreted as a disjoint union of its components and the ground type is interpreted as the set of neutral term Ne° of ground type:

$$\begin{aligned} \llbracket o \rrbracket &::= \text{Ne}^\circ \\ \llbracket \rho \rightarrow \sigma \rrbracket &::= \llbracket \rho \rrbracket \rightarrow \llbracket \sigma \rrbracket \\ \llbracket \rho + \sigma \rrbracket &::= \llbracket \rho \rrbracket + \llbracket \sigma \rrbracket \end{aligned}$$

where $\llbracket \rho \rrbracket \rightarrow \llbracket \sigma \rrbracket$ is the full function space between the set $\llbracket \rho \rrbracket$ and $\llbracket \sigma \rrbracket$, and $\llbracket \rho \rrbracket + \llbracket \sigma \rrbracket$ is the disjoint union of the set $\llbracket \rho \rrbracket$ and $\llbracket \sigma \rrbracket$.

The *interpretation of types* $\llbracket \text{Ty} \rrbracket$ of the typed lambda calculus with sum types Λ_+ is then:

$$\llbracket \text{Ty} \rrbracket = \bigcup_{\rho \in \text{Ty}} \llbracket \rho \rrbracket$$

To define the interpretation of terms we need first to define the auxiliary notion of valuation. Given a context Γ , we define a *valuation* on Γ to be a partial function $\eta : \text{Var} \rightarrow \llbracket \text{Ty} \rrbracket_\perp$ such that for $x : \rho \in \Gamma$, we have $\eta(x)$ is defined and $\eta(x) \in \llbracket \rho \rrbracket$. Given an element $a \in \llbracket \sigma \rrbracket$, we define a valuation $(\eta, y \mapsto a)$ on $\Gamma \cup \{(y : \sigma)\}$, called the extension of η by $y \mapsto a$ by,

$$(\eta, y \mapsto a)(x) ::= \begin{cases} a & \text{if } x = y, \\ \eta(x) & \text{otherwise.} \end{cases}$$

Remark. Valuation functions are partial functions represented as total functions from the set of variables Var into the set $\llbracket \text{Ty} \rrbracket_\perp = \llbracket \text{Ty} \rrbracket + \{\star\}$, i.e., the interpretation of types extended with an element \star playing the rôle of an undefined value. In the following, whenever we will use a valuation applied to some variable, this variable will belong to the domain of definition of the valuation and the result will therefore be defined. And although, strictly speaking we should do a case distinction on the result to know whether it is defined, we will consider it to be an element of $\llbracket \text{Ty} \rrbracket$.

Finally, we define the *interpretation* $\llbracket r \rrbracket_\eta$ of a term r , whenever there is a context Γ , such that $\Gamma \vdash r : \rho$ and η is a valuation on Γ , to be an element of $\llbracket \rho \rrbracket$:

$$\begin{aligned} \llbracket x \rrbracket_\eta &::= \eta(x) \\ \llbracket \lambda x^\rho. r \rrbracket_\eta(v) &::= \llbracket r \rrbracket_{\eta, x \mapsto v} \\ \llbracket rs \rrbracket_\eta &::= \llbracket r \rrbracket_\eta (\llbracket s \rrbracket_\eta) \\ \llbracket \text{in}_i r \rrbracket_\eta &::= \iota_i \llbracket r \rrbracket_\eta \\ \llbracket \delta(r, x^\rho. s, y^\sigma. t) \rrbracket_\eta &::= \begin{cases} \llbracket s \rrbracket_{\eta, x \mapsto a} & \text{if } \llbracket r \rrbracket_\eta = \iota_0 a, \\ \llbracket t \rrbracket_{\eta, y \mapsto a} & \text{if } \llbracket r \rrbracket_\eta = \iota_1 a \end{cases} \end{aligned}$$

Several Attempts

We want now to define a reify function \downarrow , from the interpretation of types to the set of terms. we will define it inductively on the type simultaneously with a function \uparrow from the (neutral) terms to the interpretation.

Let us try to define these two functions for sum types. An element of the interpretation of a sum type is either a left or a right injection, and hence the function \downarrow can be defined by:

$$\downarrow_{\rho_0 + \rho_1} \iota_i a ::= \text{in}_i \downarrow_{\rho_i} a$$

The difficulty lies in the definition of the function \uparrow . Let us look at the problem with an example:

Example 2 (the unsolvable problem). *Let us assume we want to normalize the term xy of sum type with the following typing:*

$$x : o \rightarrow o + o, y : o \vdash xy : o + o.$$

The normalization function which is given by $\downarrow \llbracket xy \rrbracket_\uparrow$, involves the function \uparrow (in a first call, it appears as a valuation). In the course of computation, we are faced with the problem of defining the result of the function \uparrow^{o+o} applied to xy :

$$\uparrow^{o+o} xy$$

According to the interpretation the result should lie in a disjoint union, but should it be a left or a right injection? The core idea is that a sensible answer is "I don't know".

Example 3 (a solution). *The very fact that we failed to choose a value for the reflection of the term xy , will guide us in a second run of the normalization algorithm. This time we begin by a case distinction on xy , and in each branch of the alternative, we memorize whether xy corresponds to a left or a right injection.*

$$\delta(xy, z. \downarrow^{xy \mapsto \text{in}_0 z} \llbracket xy \rrbracket_\uparrow, z. \downarrow^{xy \mapsto \text{in}_1 z} \llbracket xy \rrbracket_\uparrow)$$

The function $\downarrow^{xy \mapsto \text{in}_0 z}$ follows the same definition as \downarrow except that in the computation of $\downarrow^{xy \mapsto \text{in}_0 z} \llbracket xy \rrbracket_\uparrow$, a call to the function $\uparrow xy$ will always return $\text{in}_0 \uparrow z$.

What we just described can be quite naturally implemented with exceptions and environments.

Every function is evaluated within an environment which is a finite map from terms to the injection of a variable. In the first computation of the normalization function, we begin with the empty environment; this environment corresponds to our knowledge that at a certain point in the computation we are in a certain branch of a case distinction operator δ and that within this branch, the variable bound by this operator is a left or right injection.

If in a subcomputation, the function \uparrow at sum type is applied to a term n for which we do not know whether the result should be a left or right injection (i.e., the term is not in the environment), then we abort the computation by raising an exception containing this term. In subsequent computations, we use this term n as a test term for a case distinction $\delta(n, x_0.r_0, x_1.s_1)$ and in each branch, s_i , we update the environment with this term associated to the left or the right injection, $\text{in}_i x_i$, of the bound variable of the case distinction.

The Binding Variable Problem

The problem is just a bit more complicated than explained above because the NbE algorithm can generate subterms with new bound variables as in the definition of \downarrow at arrow type:

$$\downarrow_{\rho \rightarrow \sigma} f ::= \lambda x. \downarrow f(\uparrow x)$$

If we only catch exceptions at the top-level, we would then possibly catch a term with a variable which is not yet bound. Hence we need to catch exception not only at the top-level, but after the binding symbol as well.

We need two more functions for handling exceptions, the functions `catch` and `throw`. They have the expected behaviours, i.e., in an expression `throw(n)`, the function `throw` packs the term n into an exception and throws this exception. This exception is then caught in the first embracing expression of the form `catch E h` (where `throw(n)` occurs in E), i.e., the expression `catch E h` evaluate to $h n$, the application of the handler h to the content n of the exception.

Notation. We will use the notation $\epsilon : \text{Ne} \rightarrow \text{Var} + \text{Var}$ instead of $\text{Ne} \rightarrow (\text{Var} + \text{Var})_{\perp}$ to emphasize that the environment can return an undefined element. This undefined element will be handled in the algorithm below as an exception. In the same way, we will use the notation \rightarrow_{Ne} to denote a function space where the functions can throw exceptions of type Ne .

We are now in position to write a first version of our algorithm. The environment is implemented as a partial function $\epsilon : \text{Ne} \rightarrow \text{Var} + \text{Var} = \text{Ne} \rightarrow (\text{Var} + \text{Var})_{\perp}$, taking a (neutral) term n as argument and, if defined for this n , returns an injection of a variable x , $\text{in}_0 x$ or $\text{in}_1 x$ and otherwise throws an exception. The update operation $-^{n \mapsto \text{in}_i x}$ for the environment is defined by:

$$\epsilon^{n \mapsto \text{in}_i x} n' ::= \begin{cases} \text{in}_i x & \text{if } n' = n, \\ \epsilon(n') & \text{otherwise.} \end{cases}$$

Code 1 (NbE for sum types). Let $\epsilon_0 \in \text{Ne} \rightarrow (\text{Var} + \text{Var})$ be an environment. The function $\downarrow^{\rho} : (\text{Ne} \rightarrow (\text{Var} + \text{Var})) \rightarrow$

$\llbracket \rho \rrbracket \rightarrow_{\text{Ne}} \text{PNF}$ and $\uparrow^{\rho} : (\text{Ne} \rightarrow (\text{Var} + \text{Var})) \rightarrow \text{Ne} \rightarrow_{\text{Ne}} \llbracket \rho \rrbracket$ are defined simultaneously by induction on the type ρ :

$$\begin{aligned} \downarrow_{\epsilon_0}^{\circ} n &::= n \\ \downarrow_{\epsilon_0}^{\rho \rightarrow \sigma} f &::= \lambda x. \text{doACase?}(\lambda \epsilon. \downarrow_{\epsilon}^{\sigma} f(\uparrow_{\epsilon}^{\rho} x), \{x\})_{\epsilon_0} \\ &\quad \text{with } x \text{ new} \\ \downarrow_{\epsilon_0}^{\rho_0 + \rho_1} \iota_i c &::= \text{in}_i \downarrow_{\epsilon_0}^{\rho_i} c \\ \uparrow_{\epsilon_0}^{\circ} n &::= n \\ \uparrow_{\epsilon_0}^{\rho \rightarrow \sigma} n &::= \lambda v. \uparrow_{\epsilon_0}^{\sigma} (n \downarrow_{\epsilon_0}^{\rho} v) \\ \uparrow_{\epsilon_0}^{\rho + \sigma} n &::= \begin{cases} \text{let in}_i x = (\text{catch } \epsilon_0(n) \text{ throw}(n)) \text{ in} \\ \iota_i \uparrow_{\epsilon_0}^{\rho_i} x \end{cases} \end{aligned}$$

where `doACase?` : $((\text{Ne} \rightarrow (\text{Var} + \text{Var})) \rightarrow_{\text{Ne}} \text{PNF}) \times \mathcal{P}(\text{Var}) \rightarrow (\text{Ne} \rightarrow \text{Var} + \text{Var}) \rightarrow_{\text{Ne}} \text{NF}$ is defined by

$$\begin{aligned} \text{doACase?}(f, \mathbb{X})_{\epsilon_0} &::= \text{catch } (f \epsilon_0) \\ &\quad \lambda n. \text{if } \text{FV}(n) \cap \mathbb{X} = \emptyset \text{ then } \text{throw}(n) \\ &\quad \text{else } \delta \left(n, \begin{array}{l} x. \text{doACase?}(f, (\mathbb{X} \cup \{x\}))_{\epsilon_0}^{n \mapsto \text{in}_0 x} \\ x. \text{doACase?}(f, (\mathbb{X} \cup \{x\}))_{\epsilon_0}^{n \mapsto \text{in}_1 x} \end{array} \right), \\ &\quad \text{with } x \text{ new} \end{aligned}$$

Remark that in the expression `catch $\epsilon_0(n)$ throw(n)`, ϵ_0 is a partial function and hence can return an undefined value \perp . This undefined value is handled as an exception with no content, hence the handler `throw(n)` does not take any argument. The auxiliary function `doACase?` takes three arguments. Its first argument is a function taking as first argument an environment, its second argument is a set of variables, and its third argument is an environment. In the expression `doACase?(f, \mathbb{X}) ϵ_0` , the function f is evaluated in the environment ϵ_0 , and the result is either a term or an exception $\perp(n)$ containing a neutral term n . In case of an exception $\perp(n)$, the set of free variables of n is compared to \mathbb{X} , and if there is no variable in common then the exception is thrown further up, else a δ -term is created whose test term is the neutral term n and each branch is the result of the evaluation of `doACase?` for the same function f , but in an updated environment (in each branch the neutral term n is now associated either with $\text{in}_0 x$ or $\text{in}_1 x$ for a new variable x).

The condition "with x new" appearing in the function \downarrow at arrow type and in the function `doACase?` states informally that the new binder do not bind other variables than this x . This condition can be formalized by using a "fresh name" generation mechanism, for example using a reader monad [6].

Remark. In the algorithm above, the line in the function `doACase?`:

$$\text{if } \text{FV}(n) \cap \mathbb{X} = \emptyset \text{ then } \text{throw}(n)$$

ensures that if a neutral term n thrown as an exception and containing a variable x that has not been bound either by a λ -abstraction λx directly before the `doAcase?` function or by a binding x . in a case expression after this function, this term n will be thrown upper in the computation tree. This ensures that neutral terms of sum type are captured at the uppermost binding possible, just after their creation by a binding or at the top-level. This is not a trivial idea, as we could just have caught every exceptional neutral terms after the function `doAcase?`, and create a new case distinction operator at this level.

This simpler approach would still yield a correct algorithm in the sense that the result is convertible with the term given as input in the normalization function.

The advantage of creating the distinction operator as soon as possible is that it allows to further constrain the set of normal forms, and this is essential if we are after a decision algorithm. This idea of creating the distinction operator as soon as possible first appears explicitly in the works of Vincent Balat [3], and his coauthors, Roberto Di Cosmo and Marcelo Fiore [4], with an implementation with controls operators and an election of a best prompt (uppermost binding).

The normalization function for a term r is obtained by applying the function `doAcase?` to $\lambda \epsilon \downarrow_\rho^\epsilon \llbracket r \rrbracket_{\uparrow_\epsilon}$ and all free variables of r in an empty environment, i.e., the function undefined everywhere $\lambda t. \perp$.

Code 2 (Normalization function). *The normalization function $\text{nf} : \text{Tm} \rightarrow \text{NF}$ is defined by:*

$$\text{nf}(\Gamma \vdash r : \rho) ::= \text{doAcase?}(\lambda \epsilon \downarrow_\rho^\epsilon \llbracket r \rrbracket_{\uparrow_\epsilon}, \text{FV}(t))(\lambda t. \perp)$$

Toward Completeness

Although the algorithm we have just presented is sound in the sense that results of our algorithm are $\beta\eta$ -equal to terms given as argument, it is not yet complete. In particular two terms which are $\beta\eta$ -equal can be normalized to syntactically different terms. As explained in the introduction, the syntactical equality is not suitable to identify terms which only differ in the order of independent δ -terms. What we need is an equality test based on the circular conversion (conversion π_6^e to π_9^e) and the immediate simplification (is).

First, let us consider the immediate simplification conversion:

$$\delta(r, x_0.s, x_1.s) \longrightarrow_{\text{is}} s \quad (x_0, x_1 \notin \text{FV}(r)) \quad (\text{is})$$

Whether the test term r in the rule above is a left or a right injection is irrelevant, because the two branches of the alternative are anyway the same. We call such terms redundant.

The reduction (is) above erases the whole term r , and hence the variables it contains. A consequence is that this

reduction does not preserve prenormal forms because it can break the deepest λ condition (deepest λ).

Example 4. *Suppose the following term is a prenormal form,*

$$\lambda x. \delta(t\{\delta(r, x_0.s, x_1.s)/z\}, y_0.t_0, y_1.t_1)$$

with $x \in \text{FV}(r)$ but $x \notin \text{FV}(s)$ and $x \notin \text{FV}(t)$. An immediate simplification reduction leads to the following term:

$$\lambda x. \delta(t\{s/z\}, y_0.t_0, y_1.t_1)$$

but this term does no longer verify the deepest λ condition (deepest λ) because $x \notin \text{FV}(t\{s/z\})$.

To avoid these redundancies we want to define the set of normal forms as a restriction of the set of prenormal forms. But as the example above shows, this can not be achieved by merely firing the immediate simplification reductions (is) from a term in prenormal form. To define the set of normal forms, we will integrate a test of redundancy directly in the inductive definition of prenormal forms.

Among the circular conversions, let us consider the conversion π_8^e and π_9^e .

$$\begin{aligned} \delta(r, x_0.\delta(r, x_0.s_0, x_1.s_1), x_1.t) &=_{\pi_8^e} \delta(r, x_0.s_0, x_1.t) \\ \delta(r, x_0.s, x_1.\delta(r, x_0.t_0, x_1.t_1)) &=_{\pi_9^e} \delta(r, x_0.s, x_1.t_1) \end{aligned}$$

If we consider λ -terms as programs, the term s_1 in the conversion π_8^e and the term t_0 in the conversion π_9^e are superfluous, they will never be used whatever value r is evaluated to. We call such terms junk terms. And although we do not want to select a normal form between two terms which differ only in the order of independent test, we do want to eliminate such junk terms from normal forms.

We will define the normal forms as a restriction of the prenormal forms by introducing two conditions, one to avoid redundancy (redundancy freeness) and one to avoid junk (junk freeness).

These two conditions require an equality test between two occurrences of a subterm, but using syntactical equality for this equality test is not enough. For example, if the terms r and r' only differ by the order of independent case elimination (i.e., $r =_{\pi_6^e} r'$), we still want to eliminate the junk term s_1 in the term

$$\delta(r, x_0.\delta(r', x_0.s_0, x_1.s_1), x_1.t),$$

or remove the redundancy in the term

$$\delta(r, x_0.r', x_1.t) \quad (\text{if } x_0 \notin \text{FV}(s) \text{ and } x_1 \notin \text{FV}(t)).$$

This equality has indeed to be tested modulo circular conversions.

In [1], Thorsten Altenkirch and his coauthors avoid this test modulo circular conversions altogether by defining a

normal form, which identifies terms differing in the order of independent case eliminations (independent case eliminations are grouped together into a binary function from a set of neutral terms). To keep the presentation simple, we prefer a more syntactical presentation to this elegant solution, and therefore follow the way of Vincent Balat [3] and his coauthors [4] and use a test modulo circular permutative conversions.

5 Normal Forms

We now define the normal forms of our system. Eventually, the goal of designing normal forms is to obtain a set of representatives for each class of terms modulo $\beta\eta\pi^e$ equipped with a decidable equality. The considerations in the last sections lead to the following definition of normal forms.

Definition 2 (normal forms). *We define by simultaneous induction three relations $\vdash_{\text{Ne}} - : -$, $\vdash_{\text{PNF}} - : -$, and $\vdash_{\text{NF}} - : -$ between context, term and type,*

$$\frac{(x, \rho) \in \Gamma}{\Gamma \vdash_{\text{Ne}} x : \rho} \quad \frac{\Gamma \vdash_{\text{Ne}} n : \rho \rightarrow \sigma \quad \Gamma \vdash_{\text{PNF}} P : \rho}{\Gamma \vdash_{\text{Ne}} nP : \sigma}$$

$$\frac{\Gamma \vdash_{\text{Ne}} n : o}{\Gamma \vdash_{\text{PNF}} n : o} \quad \frac{\Gamma, x : \rho \vdash_{\text{NF}} N : \sigma}{\Gamma \vdash_{\text{PNF}} \lambda x.N : \rho \rightarrow \sigma}$$

$$\frac{\Gamma \vdash_{\text{PNF}} P : \rho_i}{\Gamma \vdash_{\text{PNF}} \text{in}_i P : \rho_0 + \rho_1} \quad \frac{\Gamma \vdash_{\text{PNF}} P : \rho}{\Gamma \vdash_{\text{NF}} P : \rho}$$

$$\frac{\Gamma \vdash_{\text{Ne}} n : \rho_0 + \rho_1 \quad \Gamma, x_i : \rho_i \vdash_{\text{NF}} N_i : \sigma}{\Gamma \vdash_{\text{NF}} \delta(n, x_0.N_0, x_1.N_1) : \sigma}$$

in the rule with conclusion $\Gamma \vdash_{\text{PNF}} \lambda x.N : \rho \rightarrow \sigma$ we impose the condition

$$\text{DL}(\{x\}, N), \quad (\text{deepest } \lambda)$$

where for a set of variables \mathbb{X} and normal form N , $\text{DL}(\mathbb{X}, N)$ is true if N is not a δ -term and otherwise defined by:

$$\text{DL}(\mathbb{X}, \delta(n, x_0.N_0, x_1.N_1)) ::= \mathbb{X} \cap \text{FV}(n) \neq \emptyset \wedge \text{DL}(\mathbb{X} \cup x_0, N_0) \wedge \text{DL}(\mathbb{X} \cup x_1, N_1)$$

in the rule with conclusion $\Gamma \vdash_{\text{NF}} \delta(n, x_0.N_0, x_1.N_1) : \sigma$, we impose the condition

$$\text{RF}(\delta(n, x_0.N_0, x_1.N_1)) ::= x_0 \notin \text{FV}(N_0) \wedge x_1 \notin \text{FV}(N_1) \Rightarrow N_0 \neq_{\pi^e} N_1 \quad (\text{redundancy freeness})$$

and the condition

$$\text{JF}_\emptyset(\emptyset, \delta(n, x_0.N_0, x_1.N_1)) \quad (\text{junk freeness})$$

where $\text{JF}_\Gamma(\mathbb{N}, N)$ is true if N is not a δ -term and otherwise defined by:

$$\text{JF}_\mathbb{X}(\mathbb{N}, \delta(n, x_0.N_0, x_1.N_1)) ::= \mathbb{X} \cap \text{FV}(n) = \emptyset \Rightarrow \forall n' \in \mathbb{N}, n \neq_{\pi^e} n' \wedge \text{JF}_{\mathbb{X} \cup x_1}(\mathbb{N}, N_1) \wedge \text{JF}_{\mathbb{X} \cup x_2}(\mathbb{N}, N_2)$$

The equality on normal forms $=_{\pi^e}$ is not a mere syntactical equality modulo α -conversions, because equal normal forms can differ in the order of independent test-terms (as in example 1). Independent test-terms n_0, n_1, n_2, \dots occurs in normal forms in sequences of nested case terms of the form $\delta(n_0, x_0.\delta(n_1, y_0.\delta(n_2, z_0.N_0'', z_1.N_1'), y_1.N_1'), x_1.N_1)$ with no new bound variables (e.g., if $y_0 \in \text{FV}(n_2)$, n_1 and n_2 are not independent). Our implementation is similar to the one of Vincent Balat [3]. The test of equality $=_{\pi^e}$ has to first find maximal corresponding combinations of test-terms (here a combination up to the term N_0'' would be the set $\{n_0 \mapsto \text{in}_0 x_0, n_1 \mapsto \text{in}_0 y_0, n_2 \mapsto \text{in}_0 z_0\}$) and then compare the corresponding branches. The comparison of combination and branches is as well a test of equality $=_{\pi^e}$, but on structurally smaller terms.

To produce such normal forms, the first change in the algorithm is that the new condition of redundancy freeness has to be checked where the algorithm produces a δ -term $\delta(n, x.N_0, x.N_1)$ by testing if x occurs in N_0 or N_1 and if $N_0 =_{\pi^e} N_1$. The auxiliary function `doAcase?` is changed accordingly.

The second change concerns the environment ϵ associating a neutral term to an injection of a variable x , $\text{in}_0 x$ or $\text{in}_1 x$. This environment should now be defined on the set of neutral terms up to circular permutative conversions:

$$\epsilon^{n \mapsto \text{in}_i x} s = \begin{cases} \text{in}_i x & \text{if } s =_{\pi^e} n \\ \epsilon(s) & \text{otherwise} \end{cases}$$

Code 3 (NbE for sum types). *Let $\epsilon_0 \in \text{Ne} \rightarrow (\text{Var} + \text{Var})$ be an environment. The reify function \downarrow^ρ : $(\text{Ne} \rightarrow (\text{Var} + \text{Var})) \rightarrow \llbracket \rho \rrbracket \rightarrow_{\text{Ne}} \text{PNF}$ and reflect function \uparrow^ρ : $(\text{Ne} \rightarrow (\text{Var} + \text{Var})) \rightarrow \text{Ne} \rightarrow_{\text{Ne}} \llbracket \rho \rrbracket$ are defined simultaneously by induction on the type ρ :*

$$\begin{aligned} \downarrow_{\epsilon_0}^o n &::= n \\ \downarrow_{\epsilon_0}^{\rho \rightarrow \sigma} f &::= \lambda x. \text{doAcase?}(\lambda \epsilon. \downarrow_\epsilon^\sigma f(\uparrow_\epsilon^\rho x), \{x\})_{\epsilon_0} \\ &\quad \text{with } x \text{ new} \\ \downarrow_{\epsilon_0}^{\rho_0 + \rho_1} \iota_i c &::= \text{in}_i \downarrow_{\epsilon_0}^{\rho_i} c \\ \uparrow_{\epsilon_0}^o n &::= n \\ \uparrow_{\epsilon_0}^{\rho \rightarrow \sigma} n &::= \lambda v. \uparrow_{\epsilon_0}^\sigma (n \downarrow_{\epsilon_0}^\rho v) \\ \uparrow_{\epsilon_0}^{\rho + \sigma} n &::= \begin{cases} \text{let in}_i x = (\text{catch } \epsilon_0(n) \text{ throw}(n)) \text{ in} \\ \iota_i \uparrow_{\epsilon_0}^{\rho_i} x \end{cases} \end{aligned}$$

where `doAcase?` : $((\text{Ne} \rightarrow (\text{Var} + \text{Var})) \rightarrow_{\text{Ne}} \text{PNF}) \times$

$\mathcal{P}(\text{Var}) \rightarrow (\text{Ne} \rightarrow \text{Var} + \text{Var}) \rightarrow_{\text{Ne}} \text{NF}$ is defined by

```

doAcase?(f, X)ε₀ ::= catch (f ε₀)
  λn. if FV(n) ∩ X = ∅ then throw(n)
  else let N₀ = doAcase?(f, X ∪ {x})ε₀n ↦ in₀ x in
    let N₁ = doAcase?(f, X ∪ {x})ε₀n ↦ in₁ x in
      if x ∉ FV(N₀) ∧ N₀ =πc N₁ then N₀
      else δ(n, x.N₀, x.N₁)
    with x new

```

The auxiliary function `doAcase?` has to be applied at the top-level to catch an exception containing neutral terms containing free variables of the original term.

Code 4 (Normalization function).

$$\text{nf}(\Gamma \vdash r : \rho) = \text{doAcase?}(\lambda \epsilon \downarrow_{\rho} \llbracket r \rrbracket_{\uparrow \epsilon}, \lambda t. \perp, \text{FV}(t))$$

6. Conclusion

We have presented an algorithm for deciding equality of terms in a λ -calculus with sum types relying on the simple concept of exceptions. This algorithm could serve as a basis to explore other calculi with permutative conversions, in particular it applies mutatis mutandis to the Λ_J calculus, a calculus to write proofs of the sequent calculus in natural deduction (see the the work of the author [6]). While presented here in an informal and impure functional style, (in particular the name generation, the exception and environment mechanisms need to be formalized), the algorithm has been implemented in Haskell, and is formalized with the help of monads in a pure functional style [6]. Proofs of correctness and completeness have been carried out for an NbE algorithm using a reader monad for name generation for the simply typed λ -calculus [6], the next step is obviously to extend these proofs to a NbE algorithm using exception for sum type, i.e., to prove the two claims below.

Claim 1 (Correctness).

$$r =_{\beta\eta\pi^e} \text{nf}(r)$$

Claim 2 (Completeness).

$$r =_{\beta\eta\pi^e} s \implies \text{nf}(r) =_{\pi^e} \text{nf}(s)$$

Another interesting question is raised by the relationship between exceptions and continuations. An exception can be seen as a particularly simple form of exception. Is it possible to design a sound transformation between our exception based algorithm and an optimized one based on continuations?

References

- [1] T. Altenkirch, P. Dybjer, M. Hofmann, and P. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 303–310, 2001.
- [2] T. Altenkirch and T. Uustalu. Normalization by evaluation for $\lambda^{\rightarrow,2}$. In Y. Kameyama and P. J. Stuckey, editors, *Proc. of 7th Int. Symp. on Functional and Logic Programming, FLOPS 2004 (Nara, Japan, 7–9 Apr. 2004)*, volume 2998 of *Lecture Notes in Computer Science*, pages 260–275. Springer, Berlin, 2004.
- [3] V. Balat. *Une étude des sommes fortes : isomorphismes et formes normales*. PhD thesis, Université Paris 7, 2002.
- [4] V. Balat, R. D. Cosmo, and M. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. *ACM SIGPLAN Notices*, 39(1):64–76, Jan. 2004.
- [5] V. Balat and O. Danvy. Memoization in type-directed partial evaluation. In D. Batory, C. Consel, and W. Taha, editors, *Proceedings of the 2002 ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering, GPCE 2002*, number 2487 in *Lecture Notes in Computer Science*, pages 78–92, Pittsburgh, Pennsylvania, Oct. 2002. ACM, Springer.
- [6] F. Barral. *Decidability for Non-Standard Conversion in Typed Lambda-Calculi*. PhD thesis, Ludwig-maximilians-University, Université Paul Sabatier, Munich, June 2006.
- [7] O. Danvy. Type-directed partial evaluation. In G. L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, Jan. 1996. ACM, ACM Press.
- [8] P. de Groote. Strong normalization of classical natural deduction with disjunction. In S. Abramsky, editor, *Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2001.
- [9] D. J. Dougherty and R. Subrahmanyam. Equality between functionals in the presence of coproducts. *Information and Computation*, 157(1-2):52–83, 2000.
- [10] A. Filinski. Normalization by evaluation for the computational lambda-calculus. In S. Abramsky, editor, *Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, pages 151–165. Springer, May 2001.
- [11] N. Ghani. $\beta\eta$ -equality for coproducts. In M. Dezani-Ciancaglini and G. D. Plotkin, editors, *TLCA*, volume 902 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 1995.
- [12] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, 1989.
- [13] F. Joachimski and R. Matthes. Short proofs of normalisation for the simply-typed λ -calculus, permutative conversions and Gödel’s T . *Archive for Mathematical Logic*, 42(1):59–87, 2003.
- [14] K. Nour and R. David. A short proof of the strong normalization of classical natural deduction with disjunction. *The Journal of Symbolic Logic*, 68(4):1277–1288, December 2003.

- [15] D. Prawitz. Ideas and results in proof theory. In J. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 235–307. North-Holland, Amsterdam, 1971.
- [16] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, 1996.