Keeping sums under control

Vincent Balat
Laboratoire *Preuves, Programmes et Systèmes*
CNRS – Université Paris Diderot
vincent.balat @ pps.jussieu.fr

Abstract

This paper presents a normalization tool for the λ-calculus with sum types, based on the technique of normalization by evaluation, and more precisely techniques developed by Olivier Danvy for partial evaluation, using control operators. The main characteristic of this work is that it produces a result in a canonical form. That is to say: two $\beta\eta$-equivalent terms will be normalized into (almost) identical terms. It was not the case with the usual algorithm, which could even lead to an explosion of the size of code. This canonical form is an η-long β-normal form with constraints, which captures the definition of η-long normal form for the λ-calculus without sums, and reduces drastically the η-conversion possibilities for sums. We will show how this normalizer helped us to solve a problem of characterization of type isomorphisms.

1. Introduction

Partial evaluation is a transformation of programs that generates the code of specialized versions of programs to some of their inputs. Speaking in terms of λ-calculus, one would say that a partial evaluator is a strong β-normalization tool. In 1996, Olivier Danvy introduced a method for implementing powerful partial evaluators, which is called *Type-Directed Partial Evaluation* (TDPE) [13]. It is based on the original technique of *Normalization By Evaluation*, that allows to produce the code of the normal form of a term from a value and its type, even if this value is compiled! (That is, even if we cannot destructure it).

Olivier Danvy presented TDPE for languages à la ML, and introduced a mechanism for inserting `let` instructions to respect the order of evaluation, which was necessary to preserve the observational equivalence in the case of a language with side-effects. To achieve this, he uses the control operators *shift* and *reset* (Danvy and Filinski [15, 16]). These operators were also used to handle sum types.

In this paper, we are interested in the case of normalizing simply typed λ-calculus, extended with product, unit and sum types. We target the canonical normal form we introduced in [8]. This paper is an extended version of the last part of that one. The idea consists of using TDPE to normalize λ-terms *written as ML functions*. But this problem is slightly different to the one of specializing ML programs, because we are in a world without side effects and without `let` constructs (for the language we want to normalize).

1.1. Type-Directed Partial Evaluation

Figure 1 shows a version of TDPE without `let` insertion. For detailed information on the way it works, please refer to Olivier Danvy's papers [13, 14]. Here are just a few elements.

A two-level λ-term (in Curry notation) is a term of the shape t given by the following grammar:

$$
\begin{array}{rcl}
t & ::= & \underline{d} \mid s \\
s & ::= & x \mid \lambda x.\, t \mid t \mathbin{@} t \mid \\
 & & () \mid pair(t,\, t) \mid proj_1\ t \mid proj_2\ t \mid \\
 & & in_1\ t \mid in_2\ t \mid case\,(t,\, x_1.\, t,\, x_2.\, t) \\
\underline{d} & ::= & \underline{x} \mid \underline{\lambda x}.\, t \mid t \mathbin{\underline{@}} t \mid \\
 & & \underline{()} \mid \underline{pair}(t,\, t) \mid \underline{proj_1}\ t \mid \underline{proj_2}\ t \mid \\
 & & \underline{in_1}\ t \mid \underline{in_2}\ t \mid \underline{case}\,(t,\, x_1.\, t,\, x_2.\, t)
\end{array}
$$

where x (resp. $\underline{x}$) ranges over (a countable set of) *static* (resp *dynamic*) variables. The s-terms are said to be *static*, and the d-terms to be *dynamic*. In the implementation in ML, dynamic terms are represented by data-structures, whereas static terms are the ML language itself. We use the infix operators $@$ or $\underline{@}$ for the application to allow the distinction between static and dynamic application. Terms $case$ and $\underline{case}$ are called *discriminators*.

The TDPE algorithm defines two functions indexed by types. One is called `reify` and written $\downarrow$, the other one is called `reflect` and written $\uparrow$. To normalize a static term

t of type τ, apply the function $\downarrow^\tau$ to t, then reduce the static part, to obtain a fully dynamic term, which is provably in normal form. The functions $\downarrow$ and $\uparrow$ are basically two-level η-expansions. The reduction of static parts is performed automatically by the abstract machine of the language. Control operators are used to place $case$ into the right place in the final result.

shift/reset We briefly explain the way in which *shift* and *reset* work with an example. For details, see [15, 16]. The operator *reset* is used to delimit a context of evaluation. Then *shift* abstracts this context in a function. Thus the term

$$1 + \mathit{reset}\ (\mathbf{2} + \mathit{shift}\ c.\ (3\ +\ (\boldsymbol{c}\ 4)\ +\ (\boldsymbol{c}\ 5)))$$

reduces to $1\ +\ 3\ +\ (\mathbf{2}+4)+(\mathbf{2}+5)$.

The operator *reset* delimits the context $2\ +\ \square$ (where $\square$ is a "hole"), which is abstracted into the function c, then 4 and 5 are successively inserted in this context, and the resulting expression is evaluated.

In this paper, we will use a version of TDPE we wrote for the language *Objective Caml* (see [5])[1].

We will use the following type for binary sums:

```
type ('a,'b) sum = L of 'a | R of 'b;;
```

Our normalizing function is called `residualize`. It takes as first argument the type of the term to normalize, or more precisely the pair `(reify, reflect)` associated with this type. Following Andrzej Filinski and Zhe Yang's method, we will remark that it is possible to construct the pair $(\downarrow^{\tau_1\to\tau_2}, \uparrow^{\tau_1\to\tau_2})$ from $(\downarrow^{\tau_1}, \uparrow^{\tau_1})$ and $(\downarrow^{\tau_2}, \uparrow^{\tau_2})$ by an (infix) function `**->`. Same for pairs and sums.

Thus, for example, to normalize a value `v` of type $\theta+\theta\to\theta\times\theta$ (where θ is a base type), we will write

```
# residualize
  ((sum (base, base)) **-> (prod (base, base))) v
```

Here `((prod (base, base))` is the pair $(\downarrow^{\theta\times\theta}, \uparrow^{\theta\times\theta})$ and `((sum (base, base))` is the pair $(\downarrow^{\theta+\theta}, \uparrow^{\theta+\theta})$.

The result of the normalization is pretty-printed in the *Objective Caml* syntax.

1.2. First example

To see where changes are needed in the TDPE algorithm, let us test the rezidualisation function on an example (suggested by Andrzej Filinski, and also considered in [2]). It is possible to show that for any boolean function f, one has $f\circ f\circ f = f$. Thus let us define the following function:

```
# let fff f x = f (f (f x));;
val fff : ('a -> 'a) -> 'a -> 'a = <fun>
```

Let us define the value `bool` this way:

[1] Actually it is a version of *Objective Caml* slightly modified to make possible the use of control operators.

$$
\begin{array}{rcl}
\downarrow^\theta t & = & t \\
\downarrow^1 t & = & \underline{()} \\
\downarrow^{\sigma\to\tau} t & = & \underline{\lambda x}.\ \mathit{reset}\ \downarrow^\tau (t\ @\ \uparrow^\sigma \underline{x}) \qquad (\underline{x}\ \text{fresh}) \\
\downarrow^{\tau_1\times\tau_2} t & = & \underline{pair}(\downarrow^{\tau_1}(proj_1\ t),\ \downarrow^{\tau_2}(proj_2\ t)) \\
\downarrow^{\tau_1+\tau_2} t & = & \\
 & & case\left(t,\ x_1.\ \underline{in_1}\ (\downarrow^{\tau_1} x_1),\ x_2.\ \underline{in_2}\ (\downarrow^{\tau_2} x_2)\right) \\
\uparrow^\theta \underline{e} & = & \underline{e} \\
\uparrow^1 \underline{e} & = & () \\
\uparrow^{\tau\to\sigma} \underline{e} & = & \lambda x.\ \uparrow^\sigma (\underline{e}\ @\ \downarrow^\tau x) \\
\uparrow^{\sigma_1\times\sigma_2} \underline{e} & = & pair(\uparrow^{\sigma_1}(\underline{proj_1}\ \underline{e}),\ \uparrow^{\sigma_2}(\underline{proj_2}\ \underline{e})) \\
\uparrow^{\sigma_1+\sigma_2} \underline{e} & = & \\
 & & \mathit{shift}\ c.\ \underline{case}\begin{pmatrix}\underline{e},\\ \underline{x_1}.\ \mathit{reset}\ (c\ @\ in_1\ (\uparrow^{\sigma_1}\underline{x_1})),\\ \underline{x_2}.\ \mathit{reset}\ (c\ @\ in_2\ (\uparrow^{\sigma_2}\underline{x_2}))\end{pmatrix}
\end{array}
$$

Figure 1. Type-directed normalization (without let insertion).

```
# let bool = sum (unit,unit);;
```

Figure 2 shows the result of the normalization of `fff` by TDPE. We know that `fff` $=_{\beta\eta}$ `id`, but the residualization of the identity with the same type produces code which is much more concise:

```
# let id x = x;;
val id : 'a -> 'a = <fun>
# residualize
   ((bool **-> bool) **-> (bool **-> bool)) id;;
- : normal =
(fun v0 v1 -> (match v1 with
              | L  v2 -> (match v0 (L ()) with
                          | L v4 -> L ()
                          | R v4 -> R ())
              | R  v2 -> (match v0 (R ()) with
                          | L v3 -> L ()
                          | R v3 -> R ())))
```

The first thing we notice is the size of the code generated by TDPE from `fff`, which is much more important than for the identity. They are both in β-normal form, and fortunately, it is possible to show that the two terms are η-equivalent. The goal of this paper is to get a canonical form independent of the choice of the input in a $\beta\eta$-equivalence class. It became crucial when we wanted to solve a problem concerning isomorphisms of types, which is at the origin of this work. It is explained in the next section.

```
 # residualize
     ((bool **-> bool) **-> (bool **-> bool))
     fff;;
- : normal =
(fun v0 v1 ->
  (match v1 with
  | L v2 ->
    (match v0 (L ()) with
    | L v10 -> (match v0 (L ()) with
                | L v14 -> (match v0 (L ()) with
                            | L v16 -> L ()
                            | R v16 -> R ())
                | R v14 -> (match v0 (R ()) with
                            | L v15 -> L ()
                            | R v15 -> R ()))
    | R v10 -> (match v0 (R ()) with
                | L v11 -> (match v0 (L ()) with
                            | L v13 -> L ()
                            | R v13 -> R ())
                | R v11 -> (match v0 (R ()) with
                            | L v12 -> L ()
                            | R v12 -> R ())))
  | R v2 ->
   (match v0 (R ()) with
   | L v3 -> (match v0 (L ()) with
               | L v7 -> (match v0 (L ()) with
                          | L v9 -> L ()
                          | R v9 -> R ())
               | R v7 -> (match v0 (R ()) with
                          | L v8 -> L ()
                          | R v8 -> R ()))
   | R  v3 -> (match v0 (R ()) with
               | L v4 -> (match v0 (L ()) with
                          | L v6 -> L ()
                          | R v6 -> R ())
               | R v4 -> (match v0 (R ()) with
                          | L v5 -> L ()
                          | R v5 -> R ())))))
```

Figure 2. Normalization of `fff` by TDPE.

1.3. Application to type isomorphisms

Two data types are said to be isomorphic if it is possible to convert data between them without loss of information. More formally, two types σ and τ are isomorphic if there exists a function f of type $\sigma \to \tau$ and a function g of type $\tau \to \sigma$, such that $f \circ g$ is the identity function over τ and $g \circ f$ is the identity function over σ.

Type isomorphisms provide a way not to worry about unessential details in the representation of data. They are used in functional programming to provide a means to search functions by types [17, 18, 26, 27] and to match modules by specifications [10, 3].

Searching for converters between particularly complex isomorphic types raises the problem of normalizing composite functions, in order to verify whether they are the identity function or not. Normalization by evaluation provides an elegant solution: we simply write the functions in ML and we residualize their composition.

The work presented in this paper takes its inspiration from a joint work with Roberto Di Cosmo, and Marcelo Fiore [7]. This work addresses the relations between the problem of type isomorphisms and a well-known arithmetical problem, called "Tarski's high school algebra problem" [19].

1.3.1 Tarski's high school algebra problem

Tarski asked whether the arithmetic identities taught in high school (namely: commutativity, associativity, distributivity and rules for the neutral elements and exponentiation) are complete to prove all the equations that are valid for the natural numbers. His student Martin answered this question affirmatively under the condition that one restricts the language of arithmetic expressions to the operations of product and exponentiation and the constant 1.

For arithmetic expressions with sum, product, exponentiation, and the constant 1, however, the answer is negative, witness an equation due to Wilkie that holds true in $\mathbb{N}$ but that is not provable with the usual arithmetic identities [29]. Furthermore, Gurevič has shown that in that case, equalities are not finitely axiomatizable [24]. To this end, he exhibited an infinite number of equalities in $\mathbb{N}$ such that for every finite set of axioms, one of them can be shown not to follow.

1.3.2 Tarski's high school algebra problem, type-theoretically

If one replaces sum, product, and exponentiation respectively by the sum, product, and arrow type constructors, and if one replaces the constants 0 and 1 respectively by the empty and unit types, one can restate Tarski's question as one about the isomorphisms between types built with these constructors. For types built without sum and empty types, Soloviev, and then Bruce, Di Cosmo, and Longo have shown that the axioms are exactly the same [12, 28].

Continuing the parallel with arithmetic, we studied the case of isomorphisms of types with empty and sum types [7]. We generalized Gurevič's equations for the case of equalities in $\mathbb{N}$ without constants as follows:

$$(A^u+{B_n}^u)^v\cdot({C_n}^v+{D_n}^v)^u \;=\; (A^v+{B_n}^v)^u\cdot({C_n}^u+{D_n}^u)^v \qquad (n \geq 3, \text{ odd})$$

where $A = y + x \qquad B_n = \sum_{i=0}^{n-1} x^i y^{n-i-1}$

$C_n = y^n + x^n \qquad D_n = \sum_{i=0}^{n-1} x^{2i} y^{2n-2i-2}$

We proved that these equalities hold in the world of type isomorphisms as well. We did so by exhibiting a family of functions and their inverses. Figure 3 shows one of these functions, written in *Objective Caml*, when $n = 3$. Let us

call it `f3`. The type of this term fragment is displayed at the bottom of the figure. It corresponds to $(A^u + {B_n}^u)^v \cdot (C_n{}^v + D_n{}^v)^u \ \rightarrow \ (A^v + {B_n}^v)^u \cdot ({C_n}^u + {D_n}^u)^v$, where `sum`, `*`, and `->` are type constructors for sums, products, and functions (*i.e.*, exponentiations).[2]

For such large and interlaced functions, it is rather daunting to show that composing them with their inverse yields the identity function. A normalization tool that handles sums is needed.

In the presence of sums, however, normalization is known to be a non-trivial affair [1, 8, 25]. TDPE does handle sums, but the application to this problem results in an explosion of the size of the code. Indeed, let us call `comp3` the composition of this function with its supposed inverse (in fact it an involution). The result of this residualization is presented partly in figure 4. The complete version takes approximately 1200 lines (whereas f3 takes only 52 lines) ... The question is: is it the identity?

2. A new normalizer for the lambda-calculus with sums

2.1. Normalization with sum types

What first strikes when looking at the result of the normalization of `comp3` is that many applications are computed more than once. The version of TDPE with introduction of `let` does not solve the problem. In [6], we propose to combine the `let` insertion with a mechanism of memoization, to produce a fully-lazy partial evaluator. This produces a first improvement on the size of the result.

In the case of the λ-calculus without `let`, the explanation of the behavior of TDPE is to be found in the fact that there are way too many possibilities of η-conversion for the same term. TDPE produces an η-expanded normal form, but whereas in the case without sum the η-expansion is controlled (we can speak about η-long normal form), it is much more complex with sums, and the notion of η-long normal form is not as clear.

The problem of normalization of the λ-calculus with sums is very complex. Indeed, even with rewriting techniques, we can get results completely different from the same term, depending on the order of reduction, due to the fact that it is not confluent. In fact, the only viable approach is that of reductionless normalization, as put forward for sums in [21], and further investigated in [1], where a sophisticated system is proposed to define directly normal forms, using n-ary sums. In this work, we use instead the system for binary sums we introduced in [8]. It is a new notion of normal form for the λ-calculus with sums, defined extensionally. Using the category of Grothendieck logical relations, we built a system of inference rules with *constraints* and proved that every term is $\beta\eta$-equivalent to a term of this shape.

The inference rules define the notion of *normal* terms, and impose them to be in β-normal form. The constraints are strong enough to reduce drastically the possibilities of η-conversion.

[2] In ML's type language, the type constructors for products and functions are infix, and the type constructor for sums is postfix.

```
let f3 =
fun (a1,a2) -> (
(fun u ->
  match a2 u with
  | L b -> L (fun v ->
     match a1 v with
     | L c -> c u
     | R c -> (match (c u),(b v) with
       | (L d), (L e) -> L (fst d)
       | (L d), (R e) -> R (fst e)
       | (R (L d)), (L e) -> R (snd d)
       | (R (L d)), (R e) -> L (fst d)
       | (R (R d)), (L e) -> L (fst e)
       | (R (R d)), (R e) -> R (fst d)))
  | R b -> R (fun v ->
     match a1 v with
     | L c -> (match (c u),(b v) with
       | (L d), (L e) -> L (d,(fst e))
       | (L d), (R (L e)) -> R (R (snd (snd e)))
       | (L d), (R (R e)) -> R (L ((d,fst e)))
       | (R d), (L e) -> R (L ((fst e,d)))
       | (R d), (R (L e)) -> L (fst e,(fst (snd e)))
       | (R d), (R (R e)) -> R (R (d,fst e)))
     | R c -> c u)),
(fun v ->
  match a1 v with
  | L c -> L (fun u ->
     match a2 u with
     | L b -> b v
     | R b -> (match (c u),(b v) with
       | (L d), (L e) -> L (snd e)
       | (L d), (R (L e)) -> L (d,((fst e),
                                   (fst (snd e))))
       | (L d), (R (R e)) -> R (snd e)
       | (R d), (L e) -> L (snd e)
       | (R d), (R (L e)) -> R (d,(snd (snd e)))
       | (R d), (R (R e)) -> R (snd e)))
  | R c -> R (fun u ->
     match a2 u with
     | L b -> (match (c u),(b v) with
       | (L d), (L e) -> L ((snd d),e)
       | (L d), (R e) -> R (L (fst d,(snd d,snd e)))
       | (R (L d)), (L e) -> L ((fst d),e)
       | (R (L d)), (R e) -> R (R ((snd d),e))
       | (R (R d)), (L e) -> R (L ((fst (snd e)),
                                   ((snd (snd e)),d)))
       | (R (R d)), (R e) -> R (R ((snd d),e)))
     | R b -> b v)));;

val f3 : ('v -> ('u -> ('y, 'x) sum, 'u ->
('y * 'y, ('y * 'x, 'x * 'x) sum) sum) sum) *
('u -> ('v -> ('y * ('y * 'y), 'x * ('x * 'x)) sum,
'v -> ('y * ('y * ('y * 'y)), ('y * ('y * ('x * 'x)),
'x * ('x * ('x * 'x))) sum) sum) sum) ->
('u -> ('v -> ('y, 'x) sum, 'v ->
('y * 'y, ('y * 'x, 'x * 'x) sum) sum) sum) *
('v -> ('u -> ('y * ('y * 'y), 'x * ('x * 'x)) sum,
'u -> ('y * ('y * ('y * 'y)), ('y * ('y * ('x * 'x)),
'x * ('x * ('x * 'x))) sum) sum) sum) = <fun>
```

Figure 3. Isomorphism in the case $n = 3$.

```
#  let comp3 x = f3 (f3 x);;
#  residualize ... comp3;;
- : normal =
(fun a ->
 ((fun u ->
   (match (proj1 a) u with
   | L v160 ->
     (L (fun v ->
       (match (proj2 a) v with
       | L v241 ->
         (match (proj1 a) u with
         | L v313 -> (match v313 v with
                      | L v319 -> L v319
                      | R v319 -> R v319)
         | R v313 ->
           (match v313 v with
           | L v314 -> (match v241 u with
                        | L v318 -> L (proj1 v314)
                        | R v318 -> R (proj1 v318))
           | R v314 ->
             (match v314 with
             | L v315 -> (match v241 u with
                          | L v317 -> R (proj2 v315)
                          | R v317 -> L (proj1 v315))
             | R v315 -> (match v241 u with
                          | L v316 -> L (proj1 v316)
                          | R v316 -> R (proj1 v315)))))
       | R v241 -> (match (proj1 a) u with
                   | L v242 -> (match v242 v with
                     ...
```

Figure 4. Residualization of the composition of `f3` with itself by the usual TDPE (small excerpt: the full code is 1200 lines long).

Reductionless normal forms This inference system is presented in [8]. We will just recall the main properties of these normal forms. The many possibilities of η-conversion for the λ-calculus with sums are mainly due to the η-rule for strong sums which is the following:

$$case\left(t,\ x_1.\ t'\left[{(in_1\ \ x_1)}/_{x}\right],\ x_2.\ t'\left[{(in_2\ \ x_2)}/_{x}\right]\right) = t'\left[{t}/_{x}\right]$$

(where $x_1, x_2 \notin FV(t')$)

This rule can take many different forms, and combined with the β-rules, it allows to show a lot of conversions, sometimes intuitive, but very different from each other, as presented on figure 5. These examples show that it is not easy to write an η-reduction function.

To put constraints on the term, we first define the notion of guards of a term (see [8]) as follows:

$$\mathrm{guards}(x_i.\ N_i) \stackrel{\mathrm{def}}{=} \{\ C \in \mathrm{guards}(N_i) \mid x_i \notin FV(C)\ \},$$

$$\mathrm{guards}(case\,(M,\ x_1.\ N_1,\ x_2.\ N_2)) \stackrel{\mathrm{def}}{=} \{\ M\ \} \cup \bigcup_{i=1,2} \mathrm{guards}(x_i.\ N_i)$$

$$\mathrm{guards}(t) \stackrel{\mathrm{def}}{=} \emptyset \text{ otherwise}$$

$FV(C)$ is the set of free variable of the term C.

The three constraints on the terms are the following ones:
In a term of the shape $\lambda x.\ N$:

the variable x verifies
$x \in FV(C)$ for all $C \in \mathrm{guards}(N)$ (1)

In a term of the shape $case\,(M,\ x_1.\ N_1,\ x_2.\ N_2)$:

$M \notin \bigcup_{i=1,2} \mathrm{guards}(x_i.\ N_i)$, (2)

and if $x_1 \notin FV(N_1)$ and $x_2 \notin FV(N_2)$ then $N_1 \not\approx N_2$ (3)

We write $\approx$ for the equivalence relation generated by

$$case\,(\boldsymbol{M},\ x.\ case\,(\boldsymbol{M}_1,\ x_1.\ N_1,\ x_2.\ N_2),\ y.\ N) \approx case(\boldsymbol{M}_1,\ x_1.\ case\,(\boldsymbol{M},\ x.\ N_1,\ y.\ N),\ x_2.\ case\,(\boldsymbol{M},\ x.\ N_2,\ y.\ N)$$

$$case\,(\boldsymbol{M},\ y.\ N,\ x.\ case\,(\boldsymbol{M}_1,\ x_1.\ N_1,\ x_2.\ N_2)) \approx case\boldsymbol{M}_1,\ x_1.\ case\,(\boldsymbol{M},\ y.\ N,\ x.\ N_1),\ x_2.\ case\,(\boldsymbol{M},\ y.\ N,\ x.\ N_2)$$

when $x \notin FV(M_1)$ et $x_i \notin FV(M)$ $(i = 1, 2)$

$$\frac{N_i \approx N_i' \quad (i = 1, 2)}{case\,(M,\ x_1.\ N_1,\ x_2.\ N_2) \approx case\,(M,\ x_1.\ N_1',\ x_2.\ N_2')}$$

This relation is called equality modulo commuting conversions. It says basically that if there is no problem of variables going outside the scope of their binders, it is possible to changes the order of $case$. Actually it is not easy to be more precise about this order and that is why the normal form we obtain is not "unique". The use of an n-ary $case$ would solve this issue.

The condition (3) forbids the two branches of a case to be "identical". In such case, the $case$ would be useless. The condition (2) forbids dead branches, that is when a $case$ occurs inside one of the branches of exactly the same $case$. Finally the condition (1) fixes the position of λ with respect to the $case$. For example, the term

$$\lambda x.\ case\,(t,\ x_1.\ u,\ x_2.\ v)$$

is $\beta\eta$-equivalent to this one:

$$case\,(t,\ x_1.\ \lambda x.\ u,\ x_2.\ \lambda x.\ v) \qquad (x \notin FV(t))$$

The constraint (1) says basically that the $case$ should be lifted to the highest possible place.

We obtain a definition of normal forms that captures exactly the well known definition of η-long β-normal form for the λ-calculus without sums, and where the possibilities of η-conversion are strictly limited for sums. We will call them *canonical* normal forms.

The paper about these extensional normal forms [8] follows a paper by Marcelo Fiore [20] dealing with extensional normal forms for the λ-calculus without sums. He uses the same kind of categorical concepts to define normal terms and shows that the normalization by evaluation algorithm can be extracted from this categorical view, providing a way to compute the normal forms, without rewriting. In another paper, Thorsten Altenkirch, Peter Dybjer, Martin Hofmann and Philip Scott use similar techniques for the λ-calculus with sums [1].

$$
\begin{array}{rcl}
case\,(case\,(t,\ in_1 \circ h_1,\ in_2 \circ h_2),\ f,\ g) & =_{\beta\eta} & case\,(t,\ f \circ h_1,\ g \circ h_2) \\
(\boldsymbol{f}\ \ (case\,(t,\ x_1.\ t_1,\ x_2.\ t_2))) & =_{\beta\eta} & case\,(t,\ x_1.\ (\boldsymbol{f}\ \ t_1),\ x_2.\ (\boldsymbol{f}\ \ t_2)) \\
((case\,(t,\ x_1.\ t_1,\ x_2.\ t_2))\ \ \boldsymbol{t'}) & =_{\beta\eta} & case\,(t,\ x_1.\ (t_1\ \ \boldsymbol{t'}),\ x_2.\ (t_2\ \ \boldsymbol{t'})) \\
case\,(\boldsymbol{t},\ x.\ \boldsymbol{case\,(t,\ x_1.\ t_1,\ x_2.\ t_2)},\ y.\ u) & =_{\beta\eta} & case\left(\boldsymbol{t},\ x.\ t_1\left[{}^{x}/_{x_1}\right],\ y.\ u\right) \\
case\,(\boldsymbol{t},\ x.\ case\,(\boldsymbol{t'},\ x_1.\ u_1,\ x_2.\ u_2),\ y.\ u) & =_{\beta\eta} & case\,(\boldsymbol{t'},\ x_1.\ case\,(\boldsymbol{t},\ x.\ u_1,\ y.\ u),\ x_2.\ case\,(\boldsymbol{t},\ x.\ u_2,\ y.\ u)) \\
 & & \text{if } x \notin FV(t') \text{ and } x_i \notin FV(t)\ \ (i = 1, 2)
\end{array}
$$

Figure 5. Some examples of $\beta\eta$-conversion with sum types

The concrete implementation of the technique is not obvious if we want to keep the main principle of normalization by evaluation which is to work on semantic (compiled) values to produce the code of its normal forms (see [11], that's why we can say that it acts as a decompiler). Indeed, a naive implementation of the normalization by evaluation algorithm would need to look at the shape of compiled values (which is not possible) in order to know whether to put a $case$ or not just after a λ. In this paper, we propose to solve this problem using Olivier Danvy's solution, namely the use of control operators. After each λ, we introduce a *reset* and we use *shift* to put a $case$ at this place only if needed.

TDPE's normal forms In the following, we will show that the usual TDPE does not produce our canonical normal forms, and how to transform it to achieve this goal.

Let us look at a simple example. The residualization of the following function does not satisfy the condition (1), since `(v2 v0)` does not contain the variable `v4`:

```
# let f t x g = match g x with
                | L c -> (fun y -> L y)
                | _ -> (fun y -> (g t));;
# residualize (base **-> (base **-> ((base **->
(sum (base, base))) **-> ((base **-> (sum (base,
base)))))))) f;;
- : normal =
(fun v0 v1 v2 -> (match v2 v1 with
  | L v3 -> (fun v6 -> L v6)
  | R v3 -> (fun v4 -> (match v2 v0 with
                        | L v5 -> L v5
                        | R v5 -> R v5))))
```

By observing this result of the normalization of the term `fff` (figure 2), we can see that it does not observe the constraints, because there are two "`match (v0 (L ()))`" nested (condition (2)).

2.2. Three changes

2.2.1 Remove dead branches

By quickly studying the code produced by TDPE on the function `comp3` (see figure 4), one sees immediately that many branches of the term are never reached. To solve this problem, we can use the following facts, which are consequences of the η rule for strong sums:

$$
\begin{array}{l}
case\,(\boldsymbol{t},\ x.\ \boldsymbol{case\,(t,\ x_1.\ t_1,\ x_2.\ t_2)},\ y.\ u) \\
\qquad =_{\beta\eta} case\left(\boldsymbol{t},\ x.\ t_1\left[{}^{x}/_{x_1}\right],\ y.\ u\right)
\end{array}
$$

$$
\begin{array}{l}
case\,(\boldsymbol{t},\ x.\ u,\ y.\ \boldsymbol{case\,(t,\ x_1.\ t_1,\ x_2.\ t_2)}) \\
\qquad =_{\beta\eta} case\left(\boldsymbol{t},\ x.\ u,\ y.\ t_2\left[{}^{y}/_{x_2}\right]\right)
\end{array}
$$

To apply these transformations, notice that the residual program is an abstract syntax tree built in depth-first manner, from left to right, the evaluation being done in call by value. The idea consists in maintaining a global table accounting for the conditional branches in the path from the root of the residual program to the current point of construction (see [6]). This table associates a flag (L or R) and a variable to an expression in the following way:

$$
\uparrow^{\sigma_1+\sigma_2} \underline{e} \quad = \quad \begin{cases}
in_1\ \ (\uparrow^{\sigma_1} \underline{z}) & \\
\qquad \text{if } e \text{ is globally associated to } (L,\ \underline{z}) & \\
in_2\ \ (\uparrow^{\sigma_2} \underline{z}) & \\
\qquad \text{if } e \text{ is globally associated to } (R,\ \underline{z}) & \\
\textit{shift}\ c.\ \underline{case}\begin{pmatrix} e, \\ \underline{x_1}.\ \textit{reset}\ \ in_1\ \ \uparrow^{\sigma_1} \underline{x_1}, \\ \underline{x_2}.\ \textit{reset}\ \ in_2\ \ \uparrow^{\sigma_2} \underline{x_2} \end{pmatrix} & \\
\qquad \text{otherwise} & \\
 & \\
\qquad \text{(where } \underline{x_1} \text{ and } \underline{x_2} \text{ are fresh)} &
\end{cases}
$$

If e is associated to nothing in the table, then we associate to it the pair $(L,\ x_1)$ while entering the first branch of the $case$, and $(R,\ x_2)$ while entering the second one.

The test of global association is done modulo $\approx$ (see next section).

This optimization, associated with let insertion and other memoization techniques, has already been used for building a fully lazy partial evaluator from TDPE (see [6]).

2.2.2 Forbid redundant discriminators

To enforce condition (2), we implement a test of the congruence $\approx$ of two normal terms. For that, we first write a function checking the equality of two dynamic terms modulo α-conversion, and a test of membership to free variables. These functions do not pose any theoretical problem.

Then we write a function testing the equality modulo commutative conversions of two normal terms, supposing that they both satisfy already the condition (2). We can easily prove the following lemma:

Lemma: *If N_1 and N_2 are two normal terms satisfying the condition (2) and such that $N_1 \approx N_2$ then* $\text{guards}(N_1) = \text{guards}(N_2)$.

The first step of our function can thus be the test of the equality of the two sets of guards. If they are different, the terms are not equivalent. If not, one continues according to the following method. For each guard we have two possible choices, that is to say 2^n choices (where n is the number of guards). For each one of these choices, one can easily find the branch of the term concerned, and check that it is the same one for the two terms (modulo α-equivalence). We do not need to recursively check the condition inside these branches since it is known that N_1 and N_2 satisfy already the condition (2), because TDPE builds the term in depth first manner.

2.2.3 Fix the relative positions of abstractions and discriminators

To obtain terms in the canonical normal form, we must also check the condition (1) concerning the guards of the abstractions.

For that, let us look at the example at the end of section 2.1. We want to introduce the `match (v2 v0)` above `fun v4 ...` However a *shift* always returns to the preceding *reset*. It would be convenient to be able to name each *reset* and to choose the best at the time to introduce the `match`. It is what the control operators *cupto*/*set* allow to do.

cupto*/*set The control operators *set* and *cupto* were introduced in 1998 by Carl A. Gunter, Didier Rémy and Jon G. Riecke [23]. They generalize the exceptions and the continuations (whereas *shift* and *reset* do not make it possible to code the exceptions).

For the detail of their operational semantics, see the article aforementioned. The operators *cupto* and *set* rely on the concept of *prompt*, that allows to mark the occurrences of `set`. New prompts can be created upon request. For two prompts p_1 and p_2, one can write an expression like:
$1+$*set* $\boldsymbol{p_1}$ *in* $\mathbf{2}+$*set* $\boldsymbol{p_2}$ *in* $\mathbf{3}+$*cupto* $\boldsymbol{p_1}$ *as* $\boldsymbol{c}$ *in* $(\mathbf{4} + (\boldsymbol{c}\ \mathbf{5}))$
which evalutaes to $1\ +\ 4\ +\ (\mathbf{2}\ +\ \mathbf{3} + 5)$.

Application to TDPE To use these control operators for our problem, we must create a new prompt with each dynamic $\underline{\lambda}$ created (empty set of guards). We maintain a global list associating to each prompt a set of variables. To introduce a new `case`, we look for all the free variables of its condition, and look in this list for the last prompt introduced associated with these variables. The term being built in depth first manner and from left to right, one obtains a closed term.

We thus modify the algorithm of TDPE this way:

$$\downarrow^{\sigma\to\tau} t \quad = \quad \underline{\lambda x}.\ \textit{set}\ p\ \textit{in}\ \downarrow^{\tau}\ (t\ @\ \uparrow^{\sigma}\ \underline{x})$$
($\underline{x}$ fresh variable, p new prompt)

$$\uparrow^{\sigma_1+\sigma_2}\ \underline{e} \quad = \quad \textit{cupto}\ m\ \textit{as}\ c\ \textit{in}\ \underline{case}\begin{pmatrix}\underline{e},\\ \underline{x_1}.\ \textit{set}\ m\ \textit{in}\ (c\ @\ in_1\ \ (\uparrow^{\sigma_1}\ \underline{x_1})),\\ \underline{x_2}.\ \textit{set}\ m\ \textit{in}\ (c\ @\ in_2\ \ (\uparrow^{\sigma_2}\ \underline{x_2}))\end{pmatrix}$$

where m is the best prompt for $\underline{e}$.

The "best prompt" is computed according to the rules of the canonical form. Basically, it means : "the highest possible place" whithout escaping the scope of variables. Note that like this, the algorithm can loop on certain examples. But by associating this optimization with that for the condition (2), we obtain an algorithm which always terminates. The final algorithm is presented at figure 6.

Discussion on control operators The new algorithm does not use all the power of the operators *cupto*/*set* ; in particular we don't use their ability to code the exceptions. We could thus use only a restricted version of these operators. There is for example a hierarchical version of *shift*/*reset*, making possible to have several levels of control (see Danvy-Filinski [15]). But they require to know by advance the maximum depth necessary, which is impossible in our case. After multiple discussions with Olivier Danvy, Andrzej Filinski and Didier Rémy, an implementation with *shift*/*reset* (hierarchical or not) does not seem obvious, even if it seems possible theoretically.

2.3. Results and application to isomorphisms of types

Observe the code produced by the new normalizer for the function `f` at the end of section 2.1.

$$
\begin{array}{rcl}
\downarrow^{\theta} t & = & t \\
\downarrow^{\mathbf{1}} t & = & \underline{()} \\
\downarrow^{\sigma\to\tau} t & = & \underline{\lambda x}.\ \textit{set}\ p\ \textit{in}\ \downarrow^{\tau}\ (t\ @\ \uparrow^{\sigma}\ \underline{x}) \qquad (\underline{x}\ \text{fresh variable},\ p\ \text{new prompt}) \\
\downarrow^{\tau_1\times\tau_2} t & = & \underline{pair}(\downarrow^{\tau_1}\ (proj_1\ \ t),\ \downarrow^{\tau_2}\ (proj_2\ \ t)) \\
\downarrow^{\tau_1+\tau_2} t & = & case\left(t,\ x_1.\ \underline{in_1}\ \ (\downarrow^{\tau_1}\ x_1),\ x_2.\ \underline{in_2}\ \ (\downarrow^{\tau_2}\ x_2)\right) \\
\\
\uparrow^{\theta} \underline{e} & = & \underline{e} \\
\uparrow^{\mathbf{1}} \underline{e} & = & () \\
\uparrow^{\tau\to\sigma} \underline{e} & = & \lambda x.\ \uparrow^{\sigma}\ (\underline{e}\ \underline{@}\ \downarrow^{\tau} x) \\
\uparrow^{\sigma_1\times\sigma_2} \underline{e} & = & pair(\uparrow^{\sigma_1}\ (\underline{proj_1}\ \ \underline{e}),\ \uparrow^{\sigma_2}\ (\underline{proj_2}\ \ \underline{e})) \\
\uparrow^{\sigma_1+\sigma_2} \underline{e} & = & \left\{\begin{array}{l}
\bullet\ in_1\ \ (\uparrow^{\sigma_1}\ \underline{z}) \qquad \text{if } e \text{ is globally associated to } (L,\ \underline{z}) \\
\bullet\ in_2\ \ (\uparrow^{\sigma_2}\ \underline{z}) \qquad \text{if } e \text{ is globally associated to } (R,\ \underline{z}) \\
\bullet\ \textit{cupto}\ m\ \textit{as}\ c\ \textit{in} \left|\begin{array}{rcl}
\textit{let} & n_1 & = \ \textit{set}\ m\ \textit{in}\ (c\ @\ in_1\ \ (\uparrow^{\sigma_1}\ \underline{x_1})) \\
\textit{and} & n_2 & = \ \textit{set}\ m\ \textit{in}\ (c\ @\ in_2\ \ (\uparrow^{\sigma_2}\ \underline{x_2})) \\
\textit{in} & \multicolumn{2}{l}{\left\{\begin{array}{ll}
n_1 & \text{if } \underline{x_1} \notin FV(n_1),\ \underline{x_2} \notin FV(n_2) \text{ and } n_1 \approx n_2 \\
\underline{case}\left(\underline{e},\ \underline{x_1}.\ n_1,\ \underline{x_2}.\ n_2\right) & \text{otherwise}
\end{array}\right.}
\end{array}\right. \\
\text{otherwise, where } m \text{ is the best prompt for } \underline{e}. \\
\text{If } \underline{e} \text{ is associated to nothing in the table, then we associate} \\
(L,\ \underline{x_1}) \text{ when entering the first branch of } \underline{case}, \\
\text{and } (R,\ \underline{x_2}) \text{ when entering the second.}
\end{array}\right.
\end{array}
$$

Figure 6. Optimized type-directed normalization.

```
# residualize2
   (base **-> (base **-> ((base **->
     (sum (base ,base))) **->
     ((base **-> (sum (base ,base))))))) f;;
- : normal =
(fun v0 v1 v2 ->
  match v2 v1 with
  | L v3 -> (fun v5 -> L v5)
  | R v4 -> match v2 v0 with
            | L v7 -> (fun v6 -> L v7)
            | R v8 -> (fun v6 -> R v8))
```

Now it respects the constraints. It is the same for `fff`:

```
# residualize2 ((bool **-> bool) **->
     (bool **-> bool)) fff;;
- : normal =
(fun v0 ->
 (match v0 (L ()) with
 | L v4 -> (match v0 (R ()) with
            | L v6 -> (fun v1 -> L ())
            | R v7 -> (fun v1 -> (match v1 with
                                 | L v2 -> L ()
                                 | R v3 -> R ()))))
 | R v5 -> (match v0 (R ()) with
            | L v10 -> (fun v1 -> (match v1 with
                                  | L v2 -> R ()
                                  | R v3 -> L ())))
            | R  v11 -> (fun v1 -> R ())))))
```

This time, the result is identical to the residualization of the identity (as the result is in the canonical normal form):

```
# residualize2 ((bool **-> bool) **->
     (bool **-> bool)) id;;
- : normal =
(fun v0 ->
 (match v0 (L ()) with
 | L v4 -> (match v0 (R ()) with
            | L v6 -> (fun v1 -> L ())
            | R v7 -> (fun v1 -> (match v1 with
                                 | L v2 -> L ()
                                 | R v3 -> R ()))))
 | R v5 -> (match v0 (R ()) with
            | L v8 -> (fun v1 -> (match v1 with
                                 | L v2 -> R ()
                                 | R v3 -> L ())))
            | R v9 -> (fun v1 -> R ())))))
```

Figure 7 shows the residualization of the function `comp3` with the new normalizer. Compared with the result presented at figure 4, it is approximately 48 times smaller (25 lines instead of 1200, and approximately 250 without taking

```
#  residualize2 ... comp3;;
- : normal =
(fun a -> ((fun u -> (match (proj1 a) u with
            | L v32 -> (L (fun v -> (match v32 v with
                                    | L v36 -> L v36
                                    | R v37 -> R v37)))
            | R v33 -> (R (fun v -> (match v33 v with
                 | L v50 -> (L ((proj1 v50) , (proj2 v50)))
                 | R v51 -> (match v51 with
                             | L v54 -> R (L ((proj1 v54) , (proj2 v54)))
                             | R v55 -> R (R ((proj1 v55) , (proj2 v55))))))))),
          (fun v ->
           (match (proj2 a) v with
            | L v0 -> (L (fun u -> (match (v0 u) with
                | L v4 -> L ((proj1 v4) , ((proj1 (proj2 v4)) , (proj2 (proj2 v4))))
                | R v5 -> R ((proj1 v5) , ((proj1 (proj2 v5)) , (proj2 (proj2 v5)))))))
            | R v1 -> (R (fun u -> (match v1 u with
                | L v20 -> (L ((proj1 v20) , ((proj1 (proj2 v20)) ,
                                              ((proj1 (proj2 (proj2 v20))) , (proj2 (proj2 (proj2 v20))))))))
                | R v21 -> (match v21 with
                            | L v22 -> (R (L ((proj1 v22) , ((proj1 (proj2 v22)),
                                             ((proj1 (proj2 (proj2 v22))) , (proj2 (proj2 (proj2 v22))))))))
                            | R v23 -> (R (R ((proj1 v23) , ((proj1 (proj2 v23)) ,
                                             ((proj1 (proj2 (proj2 v23))) , (proj2 (proj2 (proj2 v23))))))))))))))
```

Figure 7. Residualization of the composition of `f3` with itself, with the new algorithm.

account of the condition (3)).

2.4. Eta-reduction

The normal forms produced by the new normalizer are η-long normal forms, as presented in [8]. Contrary to the general case, it is possible to write an η-reduction function `etared` which will reduce these normal form into an identity not η-expanded.

This η-reduction function initially goes through the term depth first, locating the patterns corresponding to the η rules (up to α-equivalence), and replacing them by their reductions. Such a naive function does not do "all" the η-reduction in the general case (see the examples on figure 5). Actually it does not seem to be easy to define a notion of η-reduced form in presence of sums. This η-reduction function applied to the term of figure 4 give a result of hundreds of lines.

With the new normalizer, we obtain the identity not η-expanded:

```
# etared (residualize2 ... comp3);;
- : normal = (fun a -> a)
```

3. Conclusions

We presented in [6] an application of the optimization based on the condition (2) with insertion of `let` instructions. Thanks to the use of memo-functions, we obtain a "fully lazy" partial evaluator, which never evaluates twice the same sub-term. We presented benchmarks for the isomorphisms functions showing a great improvement on the size of the result. But the optimization presented in the present work allows to get better results without `let` introduction nor memoization!

To achieve this result, we mainly took into account the condition (3) from [8]. Notice that the modification concerning the constraint (1) may entail an increase of the size of the result, because if the $case$ is shifted higher, one part of the code will be duplicated in its two branches. Nevertheless, we think it is the only possible place to put the $case$ to obtain a canonical form.

As shown in the example of isomorphisms of types, this work has a theoretical interest. More generally, the new normalizer can be used to check $\beta\eta$-equivalence of λ-terms. It is interesting to see that these modifications of TDPE's algorithm can be re-used in the field of partial evaluation. Of course, this is valid only for a language without side-effects, since if the application of a function produces an effect, the application must occur as many times in the residual program as in the original.

Finally, this work provides a significant test-bed for control operators, since, as we mentioned at the end of section 2.2.3, it provides a realistic application which seems to really make use of the difference in expressivity between *shift*/*reset* and *cupto*/*set*.

This paper is a long version of a work published in 2004 (last part of [8]). Since then, some progress has been done on the subject. Freiric Barral also presented an algorithm of normalization by evaluation with sums for the same kind of normal forms, but rather than relying on control operators, he uses exceptions [9]. Sam Lindley solves the same normalization problem using a rewriting system [25], whereas Thorsten Altenkirch and Tarmo Uustalu are using decision trees [2].

4. Acknowledgments

To Marcelo Fiore and Roberto Di Cosmo who are at the origin of this work, to Olivier Danvy who made me discover TDPE, to Andrzej Filinski and Didier Rémy for interesting discussion about control operators, and to Xavier Leroy for the `call/cc` for *Objective Caml*.

References

[1] T. Altenkirch, P. Dybjer, M. Hofmann, and P. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, Boston, Massachusetts, June 2001. IEEE Computer Society Press.

[2] T. Altenkirch and T. Uustalu. Normalization by evaluation for $\lambda^{\rightarrow 2}$. In *Functional and Logic Programming*, number 2998 in LNCS, 2004.

[3] M.-V. Aponte and R. Di Cosmo. Type isomorphisms for module signatures. In *Programming Languages Implementation and Logic Programming (PLILP)*, volume 1140 of *LNCS*. Springer-Verlag, 1996.

[4] V. Balat. *Une étude des sommes fortes : isomorphismes et formes normales*. PhD thesis, PPS, Université Paris VII – Denis Diderot, Paris, France, 2002.

[5] V. Balat and O. Danvy. Strong normalization by type-directed partial evaluation and run-time code generation. In X. Leroy and A. Ohori, editors, *Proceedings of the Second International Workshop on Types in Compilation*, number 1473 in LNCS, Kyoto, Japan, 1998. Springer-Verlag.

[6] V. Balat and O. Danvy. Memoization in type-directed partial evaluation. In *ACM SIGPLAN conference on Generative Programming and Component Engineering (GCSE/SAIG)*, number 2487 in LNCS, Pittsburgh, USA, 2002.

[7] V. Balat, R. Di Cosmo, and M. Fiore. Remarks on isomorphisms in typed lambda calculi with empty and sum types. In G. D. Plotkin, editor, *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, Copenhagen, Denmark, July 2002. IEEE Computer Society Press.

[8] V. Balat, R. Di Cosmo, and M. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *Proceedings of the 31st Annual ACM Symposium on Principles of Programming Languages (POPL'04)*, volume 39, Venice, Italy, 2004. ACM.

[9] F. Barral. *Decidability for Non-Standard Conversions in Typed Lambda-Calculus*. Thèse de doctorat, Université Paul Sabatier, Toulouse, France, avril 2008.

[10] G. Barthe and O. Pons. Type isomorphisms and proof reuse in dependent type theory. In F. Honsell and M. Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001*, number 2030 in LNCS, Genova, Italy, 2001. Springer-Verlag.

[11] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In G. Kahn, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, Amsterdam, The Netherlands, 1991. IEEE Computer Society Press.

[12] K. Bruce, R. Di Cosmo, and G. Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2(2):231–247, 1992.

[13] 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*, St. Petersburg Beach, Florida, 1996. ACM.

[14] O. Danvy. Type-directed partial evaluation. In J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in LNCS, Copenhagen, Denmark, 1998. Springer-Verlag.

[15] O. Danvy and A. Filinski. Abstracting control. In M. Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, 1990. ACM.

[16] O. Danvy and A. Filinski. Representing control, a study of the cps transformation. In *Mathematical Structures in Computer Science*, number 2(4), 1992.

[17] R. Di Cosmo. Deciding type isomorphisms in a type assignment framework. *Journal of Functional Programming*, 3(3), 1993.

[18] R. Di Cosmo. *Isomorphisms of types: from λ-calculus to information retrieval and language design*. Birkhauser, 1995.

[19] J. Doner and A. Tarski. An extended arithmetic of ordinal numbers. *Fundamenta Mathematica*, 65, 1969.

[20] M. Fiore. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *4th International Conference on Principles and Practice of Declarative Programming (PPDP 2002)*. ACM, 2002.

[21] M. Fiore and A. Simpson. Lambda-definability with sums via Grothendieck logical relations. In *Typed Lambda Calculus and Applications*, number 1581 in LNCS. Springer-Verlag, 1999.

[22] C. A. Gunter, D. Rémy, and J. G. Riecke. A generalization of exceptions and control in ML. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, June 1995.

[23] C. A. Gunter, D. Rémy, and J. G. Riecke. Return types for functional continuations. unpublished, a preliminary version appeared as [22], 1998.

[24] R. Gurevič. Equational theory of positive numbers with exponentiation. *Proceedings of the American Mathematical Society*, 94(1), 1985.

[25] S. Lindley. Extensional rewriting with sums. In S. R. D. Rocca, editor, *TLCA*, volume 4583 of *LNCS*. Springer, 2007.

[26] M. Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1), 1991.

[27] C. Runciman and I. Toyn. Retrieving re-usable software components by polymorphic type. *Journal of Functional Programming*, 1(2), 1991.

[28] S. V. Soloviev. The category of finite sets and cartesian closed categories. *Journal of Soviet Mathematics*, 22(3), 1983.

[29] A. J. Wilkie. On exponentiation – a solution to Tarski's high school algebra problem. *Quaderni di Matematica*, 2001. Mathematical Institute, University of Oxford.