Keeping sums under control

Vincent Balat
INRIA - Università di Genova

January 2004 – Preliminary version

Abstract

In a recent paper [31], I presented with Marcelo Fiore and Roberto Di Cosmo a new normalisation tool for the λ-calculus with sum types, based on the technique of normalisation by evaluation, and more precisely on techniques developped by Olivier Danvy for partial evaluation, using control operators. The main characteristic of this tool is that it produces a result in a canonical form we introduced. That is to say: two $\beta\eta$-equivalent terms will be normalised into (almost) identical terms. It was not the case with the traditional 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 capture the definition of η-long normal form for the λ-calculus without sums, and reduces drastically the η-conversion possibilities for sums.

The present paper recall the definition of these normal forms and the normalisation algorithm, and shows how it is possible to use these tools to solve a problem of characterization of type isomorphisms. Indeed, the canonical form allowed to find the complicated counter-examples we exhibited in another work [6], that proves that type isomorphisms in the λ-calculus with sums are not finitely axiomatisable. What's more, when proving that these terms are isomorphisms, the new partial evaluation algorithm avoids an explosion of the size of the term that arises with the old one.

Keywords: Typed lambda calculus, Strong sums, Type isomorphisms, Normalisation, Type-Directed Partial Evaluation, *Objective Caml*.

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 β-normalisation tool. In 1996, Olivier Danvy introduced a simple method for implementing powerful partial evaluators, which is called *Type-Directed Partial Evaluation* (TDPE) [10]. It 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). It is based on a more general technique known as *Normalisation By Evaluation*, first introduced by Berger and Schwichtenberg [8]

for the simply typed lambda calculus. They presented it as an inverse to the evaluation function, mapping a semantic value into a syntactic one in normal form. Since then, NBE has been the subject of investigation in many domains: logic, type theory, category theory, partial evaluation (see, *e.g.*, [12]).

Olivier Danvy presented TDPE for languages à la ML, and introduced a mechanism of insertion of `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, [13, 14]). These operators were also used to take into account sum types.

In this paper, we are interested in the case of normalising simply typed λ-calculus, extended with product, unit and sum types. The idea consists of using TDPE to normalise 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 normalise).

1.1 Typed lambda calculus with sums

We recall the syntax and categorical semantics of the simply typed lambda calculus with binary products, unit and binary sums. For details see [25].

The set of types has a (countable) set of base types and one type constant `1` (the unit type), and is closed under the formation of product, function, and sum type constructors. Formally, types are defined by the following grammar:

$$\begin{array}{rcll} \tau & ::= & \theta & \text{(Base types)} \\ & | & \mathtt{1} & \text{(Unit type)} \\ & | & \tau_1 \times \tau_2 & \text{(Product types)} \\ & | & \tau_1 \to \tau_2 & \text{(Function types)} \\ & | & \tau_1 + \tau_2 & \text{(Sum types)} \end{array}$$

The raw terms of the calculus are defined by the following grammar:

$$\begin{array}{rcll} t & ::= & x & \text{(Variables)} \\ & | & \langle\rangle & \text{(Unit)} \\ & | & \mathrm{pair}\langle t_1, t_2\rangle & \text{(Pairing)} \\ & | & \pi_1(t) & \text{(First projection)} \\ & | & \pi_2(t) & \text{(Second projection)} \\ & | & \lambda\, x.\, t & \text{(Abstraction)} \\ & | & t_1 \mathbin{@} t_2 & \text{(Application)} \\ & | & \iota_1(t) & \text{(First injection)} \\ & | & \iota_2(t) & \text{(Second injection)} \\ & | & \delta(t, x_1.\, t_1, x_2.\, t_2) & \text{(Discriminator)} \end{array}$$

where x ranges over (a countable set of) variables.

The unit, pairing, and abstraction are respectively the term constructors for the unit, product, and function types; whilst the projections and application are respectively the term destructors for the product and function types.

The term constructors for sum types are given by the injections; whilst the discriminator is the term destructors for sum types, allowing definitions by cases.

The abstraction and discriminator are binding operators; $\lambda\, x : \tau.\, t$ binds the free occurrences of x in t, and $\delta(t, x_1.\ t_1, x_2.\ t_2)$ binds the free occurrences of x_i in t_i ($i = 1, 2$). The notions of free and bound variables are standard, and terms are identified up to alpha conversion

As usual we consider typing contexts as lists of type declarations for distinct variables, and say that a term t has type τ in the context Γ if the judgement $\Gamma \vdash t : \tau$ is derivable from the rules of Figure 1.

$$\frac{}{\Gamma,\ x : \tau,\ \Gamma' \vdash x : \tau}$$

$$\frac{}{\Gamma \vdash \langle\rangle : 1} \qquad \frac{\Gamma \vdash t_i : \tau_i \quad (i = 1, 2)}{\Gamma \vdash \mathrm{pair}\langle t_1, t_2\rangle : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i(t) : \tau_i}\ (i = 1, 2)$$

$$\frac{\Gamma,\ x : \tau_1 \vdash t : \tau}{\Gamma \vdash \lambda\, x.\, t : \tau_1 \to \tau} \qquad \frac{\Gamma \vdash t : \tau_1 \to \tau \qquad \Gamma \vdash t_1 : \tau_1}{\Gamma \vdash t\ @\ t_1 : \tau}$$

$$\frac{\Gamma \vdash t : \tau_i}{\Gamma \vdash \iota_i(t) : \tau_1 + \tau_2}\ (i = 1, 2) \qquad \frac{\Gamma \vdash t : \tau_1 + \tau_2 \qquad \Gamma,\ x_i : \tau_i \vdash t_i : \tau \quad (i = 1, 2)}{\Gamma \vdash \delta(t, x_1.\ t_1, x_2.\ t_2) : \tau}$$

Figure 1: Typing rules.

Finally, we impose the standard notion of equality on terms, including the sum extensionality axiom, as detailed in Figure 2.

1.2 Type-Directed Partial Evaluation with sums

We show how to build a normalisation algorithm based on Type-Directed Partial Evaluation that puts terms in the normal form of Section **??**. In fact, we use a version of TDPE written for the language *Objective Caml* (see [4]) slightly modified to allow the use of certain powerful control operators.

An interesting point of this work is that the optimisations we introduce will be usable in some other cases of partial evaluation. Here, however, we are only concerned in normalising functional programs corresponding to terms in the typed lambda calculus with binary sums with respect to the equational theory of the calculus. In particular, note that the normalisation of a program may have a different observational semantics (within the programming language that is) than the original program; as, for instance, the evaluation order may not be preserved.

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash t = t : \tau} \qquad \frac{\Gamma \vdash t = t' : \tau}{\Gamma \vdash t' = t : \tau} \qquad \frac{\Gamma \vdash t_1 = t_2 : \tau \qquad \Gamma \vdash t_2 = t_3 : \tau}{\Gamma \vdash t_1 = t_3 : \tau}$$

$$\frac{\Gamma \vdash t : 1}{\Gamma \vdash t = \langle\rangle : 1}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \pi_i \,\mathrm{pair}\langle t_1, t_2\rangle = t_i : \tau_i}\ (i = 1, 2) \qquad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash t = \mathrm{pair}\langle \pi_1(t), \pi_2(t)\rangle : \tau_1 \times \tau_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau \qquad \Gamma \vdash t_1 : \tau_1}{\Gamma \vdash (\lambda x.\, t)(t_1) = t\big[{}^{t_1}\!/_{x}\big] : \tau} \qquad \frac{\Gamma \vdash t : \tau_1 \to \tau}{\Gamma \vdash t = \lambda x.\, t(x) : \tau_1 \to \tau}\ (x \notin \mathrm{FV}(t))$$

$$\frac{\Gamma \vdash t : \tau_1 \to \tau \qquad \Gamma \vdash t_1 = t_1' : \tau_1}{\Gamma \vdash t(t_1) = t(t_1') : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash t = t' : \tau}{\Gamma \vdash \lambda x.\, t = \lambda x.\, t' : \tau_1 \to \tau}$$

$$\frac{\Gamma \vdash t : \tau_j \qquad \Gamma, x_i : \tau_i \vdash t_i : \tau \quad (i = 1, 2)}{\Gamma \vdash \delta(\iota_j(t), x_1.t_1, x_2.t_2) = t_j\big[{}^{t}\!/_{x_j}\big] : \tau}\ (j = 1, 2)$$

$$\frac{\Gamma \vdash t : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 + \tau_2 \vdash t' : \tau}{\Gamma \vdash \delta(t, x_1.t'\big[{}^{\iota_1(x_1)}\!/_{x}\big], x_2.t'\big[{}^{\iota_2(x_2)}\!/_{x}\big]) = t'\big[{}^{t}\!/_{x}\big] : \tau}$$

Figure 2: Equational theory of the typed lambda calculus with sums.

In the following section, I will recall the TDPE algorithm. Then I will show some examples of uses to normalise λ-terms. We will see for example that if we use it to check some complicated type isomorphisms, it produces an explosion of the size of the term. Then I will show how to adapt the algorithm to produce a canonical normal form, and apply this to these type isomorphisms examples.

2 The original TDPE

We recall the basic elements of the original TDPE algorithm. For details see [10, 11].

NBE is based on an η-expansion of the term using a two-level language, which in our case is

defined as follows:

$$
\begin{array}{rcll}
t & ::= & s & \text{(Static terms)} \\
 & | & d & \text{(Dynamic terms)} \\
 & & & \\
s & ::= & x & \\
 & | & \langle\rangle \mid \mathrm{pair}\langle t_1,\, t_2\rangle \mid \pi_1(t) \mid \pi_2(t) & \\
 & | & \lambda\, x.\, t \mid t_1 \mathbin{@} t_2 & \\
 & | & \iota_1(t) \mid \iota_2(t) \mid \delta\,(t,\, x_1.\, t_1,\, x_2.\, t_2) & \\
 & & & \\
d & ::= & \underline{x} & \\
 & & \underline{\langle\rangle} \mid \underline{\mathrm{pair}}\langle t_1,\, t_2\rangle \mid \underline{\pi_1}(t) \mid \underline{\pi_2}(t) & \\
 & | & \underline{\lambda}\,\underline{x}.\, t \mid t_1 \mathbin{\underline{@}} t_2 & \\
 & | & \underline{\iota_1}(t) \mid \underline{\iota_2}(t) \mid \underline{\delta}\,(t,\, x_1.\, t_1,\, x_2.\, t_2) &
\end{array}
$$

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 implementations, dynamic terms are often represented by data structures, whereas static terms are values of the language itself.

$$
\begin{array}{rcl}
\downarrow^{\theta} V & = & V \qquad (\theta \text{ a base type}) \\
\downarrow^{1} V & = & \underline{\langle\rangle} \\
\downarrow^{\sigma\to\tau} V & = & \text{let } \underline{x} \text{ be a fresh variable in } \underline{\lambda}\,\underline{x}.\ \mathsf{reset}(\downarrow^{\tau} (V \mathbin{@} \uparrow^{\sigma} \underline{x})) \\
\downarrow^{\tau_1\times\tau_2} V & = & \underline{\mathrm{pair}}\langle\downarrow^{\tau_1} (\pi_1(V)),\ \downarrow^{\tau_2} (\pi_2(V))\rangle \\
\downarrow^{\tau_1+\tau_2} V & = & \delta\left(V,\ x_1.\ \underline{\iota_1}(\downarrow^{\tau_1} x_1),\ x_2.\ \underline{\iota_2}(\downarrow^{\tau_2} x_2)\right) \\
 & & \\
\uparrow^{\theta} M & = & M \qquad (\theta \text{ a base type}) \\
\uparrow^{1} M & = & \langle\rangle \\
\uparrow^{\tau\to\sigma} M & = & \lambda\, x.\, \uparrow^{\sigma} (M \mathbin{\underline{@}} \downarrow^{\tau} x) \\
\uparrow^{\sigma_1\times\sigma_2} M & = & \mathrm{pair}\langle\uparrow^{\sigma_1} (\underline{\pi_1}(M)),\ \uparrow^{\sigma_2} (\underline{\pi_2}(M))\rangle \\
\uparrow^{\sigma_1+\sigma_2} M & = & \text{let } \underline{x}_1 \text{ and } \underline{x}_2 \text{ be fresh variables} \\
 & & \text{in shift } c.\ \underline{\delta}(\ M\ ,\ \underline{x}_1.\ \mathsf{reset}(c \mathbin{@} \iota_1(\uparrow^{\sigma_1} \underline{x}_1))\ ,\ \underline{x}_2.\ \mathsf{reset}(c \mathbin{@} \iota_2(\uparrow^{\sigma_2} \underline{x}_2))\ )
\end{array}
$$

Figure 3: Type-directed partial evaluation without let insertion.

The TDPE algorithm without let insertion is presented in Figure 3. It inductively defines two functions for each type. One, written $\downarrow$, is called `reify` and the other one, written $\uparrow$, is called `reflect`. The functions $\downarrow$ and $\uparrow$ are basically two-level η-expansions.

To normalise a static value V of type τ, first apply the function $\downarrow^\tau$ to V, and then reduce the static part, obtaining a fully dynamic term in normal form. The reduction of static parts is performed automatically by the abstract machine of the programming language. The control operators shift and reset are used to place δ in the right place in the final result.

Shift and reset. We briefly explain the way in which shift and reset work with an example. For details see [13, 14].

The operator reset is used to delimit a context of evaluation, and shift abstracts this context in a function. Thus the term

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

reduces to $1\ +\ 3\ +\ (\mathbf{2} + 4) + (\mathbf{2} + 5)$. Indeed, the operator reset delimits the context $2\ +\ \square$, which is abstracted into the function c; the values 4 and 5 are successively inserted in this context and the resulting expression is evaluated.

In this paper, I will use a version of TDPE we wrote for the language *Objective Caml* (see [4]). Actually it is a version of *Objective Caml* slightly modified to make possible the use of control operators.

I will use the following type for binary sums:

```
type ('a,'b) sum = Left of 'a | Right of 'b;;
```

Our normalising function is called `residualise`. It takes as first argument the type of the term to normalise, 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

$$\langle \downarrow^{\tau_1 \to \tau_2}\ ,\ \uparrow^{\tau_1 \to \tau_2} \rangle$$

from $\langle \downarrow^{\tau_1}\ ,\ \uparrow^{\tau_1} \rangle$ and $\langle \downarrow^{\tau_2}\ ,\ \uparrow^{\tau_2} \rangle$ by an (infix) function `**->`. Same for pairs and sums.

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

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

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

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

3 Normalising λ-calculus with sums using TDPE

To see where changes are needed in the TDPE algorithm, let us test the rezidualisation function on an example suggested by Andrzej Filinski.

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` in the following way:

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

Figure 4 shows the result of the normalisation of `fff` by TDPE.

We know that `fff` $=_{\beta\eta}$ `id`, but the residualisation of the identity with the same type produces code which is much more concise:

```
 # let id x = x;;
val id : 'a -> 'a = <fun>
 # residualise ((bool **-> bool) **-> (bool **-> bool)) id;;
- : Controls.ans =
(fun v0 v1 ->
   (match v1 with
    | Left  v2 -> (match (v0 (Left ())) with
                   | Left  v4 -> (Left ())
                   | Right  v4 -> (Right ()))
    | Right  v2 -> (match (v0 (Right ())) with
                   | Left  v3 -> (Left ())
                   | Right  v3 -> (Right ()))
   )
)
```

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 issue 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.

3.1 Application to isomorphisms of types

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 [16, 17, 18, 26, 27, 28, 29] and to match modules by specifications [7, 15, 2].

Searching for converters between particularly complex isomorphic types raises the problem of normalising composite functions, in order to verify whether they are the identity function or

not. Normalisation by evaluation provides an elegant solution: we simply write the functions in ML and we residualise their composition.

The work presented in this paper takes its inspiration from a recent joint work with Roberto Di Cosmo, and Marcelo Fiore [6]. 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].

3.1.1 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 [32]. 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.

3.1.2 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 exactly the same axioms are obtained [9, 30].

Continuing the parallel with arithmetic, we studied the case of isomorphisms of types with empty and sum types [6]. 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

$$
\begin{aligned}
A &= y + x \\
B_n &= y^{n-1} + xy^{n-2} + x^2y^{n-3} + \ldots + x^{n-2}y + x^{n-1} \\
&= \textstyle\sum_{i=0}^{n-1} x^i y^{n-i-1} \\
C_n &= y^n + x^n \\
D_n &= y^{2n-2} + x^2y^{2n-4} + x^4y^{2n-6} \ldots + x^{2n-4}y^2 + x^{2n-2} \\
&= \textstyle\sum_{i=0}^{n-1} x^{2i} y^{2n-2i-2}
\end{aligned}
$$

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 5 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 `'a` corresponds to v, `'b` corresponds to u, `'c` corresponds to y, `'d` corresponds to x, and furthermore `sum`, `*`, and `->` are type constructors for sums, products, and functions (*i.e.*, exponentiations).[1]

For such large and interlaced functions, it is rather daunting to show that composing them with their inverse yields the identity function. A normalisation tool that handles sums is needed. In the presence of sums, however, normalisation is known to be a non-trivial affair [1]. Type-directed partial evaluation 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 residualisation is presented partly in figure 6. The complete version takes approximately 1200 lines (whereas f3 takes only 52 lines) ... The question is: is it the identity?

4 A new normaliser for the lambda-calculus with sums

4.1 Normalisation in the presence of sum types

What first strikes when looking at the result of the normalisation 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 [5], 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

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

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 normalisation 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 normalisation, 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, I use instead the system for binary sums we introduced in [31]. 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.

Reductionless normal forms This inference system presented in [31] is shown on figure 7.

Let us 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:

$$\delta(t,\ x_1.\ t'\big[(\iota_1(x_1))/x\big],\ x_2.\ t'\big[(\iota_2(x_2))/x\big]) = t'\big[t/x\big]$$

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

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 8. These examples show that it is not easy to write an η-reduction function.

NB : In this figure (and in the whole paper) some parts of terms are written in bold font for visual distinction. It has obviously no semantic significance.

To put constraints on the term, we first define the notion of guards of a term (see [31]) as follows:

$$\mathrm{Guards}(x_i.\ N_i) \stackrel{\mathrm{def}}{=} \{\ C \in \mathrm{Guards}(N_i) \parallel x_i \notin \mathrm{FV}(C)\ \}$$

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

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

$\mathrm{FV}(C)$ 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 $x \in \mathrm{FV}(C)$ for all $C \in \mathrm{Guards}(N)$ (**A**)

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

$M \notin \bigcup_{i=1,2} \text{Guards}(x_i.\ N_i)$, (**B**)

and if $x_1 \notin \text{FV}(N_1)$ and $x_2 \notin \text{FV}(N_2)$ then $N_1 \not\approx N_2$ (**C**)

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

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

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

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

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

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 δ. 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 δ would solve this issue.

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

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

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

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

$(x \notin \text{FV}(t))$

The constraint (**A**) says basically that the δ 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. I will call them *canonical* normal forms.

The paper about these extensional normal forms [31] 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 normalisation by evaluation algorithm can be extracted from this categorical view, providing a way to compute the normal forms, without rewriting. In another recent paper, Thorsten Altenkirch, Peter Dybjer, Martin Hofmann and Philip Scott try to do the same thing for the λ-calculus with sums [1]. But the concrete implementation of the technique is not obvious if we want to keep the main principle of normalisation by evaluation which is to work on semantic (compiled) values to produce the code of its normal forms (see [8], that's why we can say that it acts as a decompiler). Indeed, a naive implementation of the normalisation 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 δ or not just after a λ.

In this paper, I 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 δ at this place only if needed.

TDPE's normal forms In the following, I will show that the traditional 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 residualisation of the following function does not satisfy the condition (**A**), since `(v2 v0)` does not contain the variable `v4`:

```
# let f t x g = match g x with
                | Left c -> (fun y -> Left y)
                | _ -> (fun y -> (g t));;

# residualise (base **-> (base **-> ((base **-> (sum (base, base))) **->
((base **-> (sum (base, base))))))) f;;
- : Controls.ans =
(fun v0 v1 v2 -> (match (v2 v1) with
  | Left  v3 -> (fun v6 -> (Left v6))
  | Right  v3 -> (fun v4 -> (match (v2 v0) with
                   | Left  v5 -> (Left v5)
                   | Right  v5 -> (Right v5)))))
```

By observing this result of the normalisation of the term `fff` (figure 4), we can see that it does not observe the constraints, because there are two `match (v0 (Left ()))` nested (condition (**B**)).

The original TDPE algorithm without let insertion produces terms following the inference system of Figure 7 without taking into account the side conditions (**A**), (**B**), (**C**) there in.

In the following, we propose three modifications of TDPE to take them into account.

4.1.1 Remove dead branches

To ensure the condition (**B**) we will use the following derivable equations:

$$\delta(t,\ x.\ \delta(t,\ x_1.\ t_1,\ x_2.\ t_2),\ y.\ t_0) \ =\ \delta(t,\ x.\ t_1[x/x_1],\ y.\ t_0)$$

$$\delta(t,\ x.\ t_0,\ y.\ \delta(t,\ x_1.\ t_1,\ x_2.\ t_2)) \ =\ \delta(t,\ x.\ t_0,\ y.\ t_2[y/x_2])$$

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. This table associates a flag (L or R) and

a variable to an expression in the following way:

$$
\begin{array}{l}
\uparrow^{\sigma_1+\sigma_2} M \;=\; \\
\quad \mathsf{if}\ M \text{ is globally associated to (L, } \underline{z}\text{) modulo } \approx \\
\quad \mathsf{then}\ \ \iota_1(\uparrow^{\sigma_1} \underline{z}) \\
\quad \mathsf{else}\ \ \mathsf{if}\ M \text{ is globally associated to (R, } \underline{z}\text{) modulo } \approx \\
\qquad \mathsf{then}\ \ \iota_2(\uparrow^{\sigma_2} \underline{z}) \\
\qquad \mathsf{else}\ \ \mathsf{shift}\ c. \\
\qquad\qquad \mathsf{let}\ \ \underline{x}_1 \text{ and } \underline{x}_2 \text{ be fresh variables,} \\
\qquad\qquad\quad \text{associate } M \text{ to } (L, \underline{x}_1) \text{ while computing} \\
\qquad\qquad\qquad n_1 = \mathsf{reset}(\iota_1(\uparrow^{\sigma_1} \underline{x}_1)), \\
\qquad\qquad\quad \text{associate } M \text{ to } (R, \underline{x}_2) \text{ while computing} \\
\qquad\qquad\qquad n_2 = \mathsf{reset}(\iota_2(\uparrow^{\sigma_2} \underline{x}_2)), \\
\qquad\qquad \mathsf{in}\ \ \underline{\delta}(\, M\,,\ \underline{x}_1.\, n_1,\ \underline{x}_2.\, n_2)
\end{array}
$$

(Note that the test of global association is done modulo $\approx$; this is explained in the next section.)

This optimisation, associated with let insertion and other memoization techniques, has been used for building a fully lazy partial evaluator from TDPE; see [5].

4.1.2 Forbid redundant discriminators

To enforce the condition (**C**), we write a test of membership of free variables and implement a test of the congruence $\approx$ of two normal terms. There are different ways in which to implement this latter test. One method is to define, in a mutually recursive fashion, three tests $\approx_{\mathcal{M}_0}$, $\approx_{\mathcal{N}_0}$, and $\approx_{\mathcal{N}}$ that respectively test the equivalence between pure neutral terms, pure normal terms, and normal terms along the following lines.

- The test $\approx_{\mathcal{M}_0}$ is done by structural recursion, using the test $\approx_{\mathcal{N}}$ in the case of applications.
- The test $\approx_{\mathcal{N}_0}$ is done by structural recursion, using the test $\approx_{\mathcal{N}}$ in the case of abstractions.
- The test $N \approx_{\mathcal{N}} N'$ inspects the set of paths p given by all possible branchings in discriminators containing the guards of N, and collects the sequence of guards together with the end pure normal form N_p. For each of these paths p, it proceeds according to the following sub-test: if N' is a pure normal term then check whether $N_p \approx_{\mathcal{N}_0} N'$, otherwise, for N' of the form $\delta(M', x.\, N'_1, y.\, N'_2)$, there are three possibilities: if M' is in the path p up to $\approx_{\mathcal{M}_0}$ and the path branches left (resp. right) the sub-test is repeated for N'_1 (resp. N'_2) instead of N', however, if M' is not in the path p up to $\approx_{\mathcal{M}_0}$, the sub-test is repeated for both N'_1 and N'_2 instead of N', succeeding if both of these sub-tests do.

Note that condition (**C**) does not need to be checked recursively within the branches of the discriminator; since, as TDPE builds the normal form in depth-first manner, it is known that each branch satisfies it.

4.1.3 Fix the relative positions of abstractions and discriminators

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

For that, let us look at the example in (4.1). We want to introduce the $\underline{\delta}(\underline{g} \ \underline{@} \ \underline{t}, \ \ldots)$ above $\underline{\lambda y}$ ... However a shift always returns to the preceding reset. Thus, it would be necessary to be able to name each reset and to choose the best one at the time of introducing the $\underline{\delta}$. This is what the control operators cupto/set, introduced in [22], allow us to do.

Set and cupto. The control operators set and cupto are very powerful, and generalise exceptions and continuations. Here we give the idea of how they work on an example. For details see [22, 23].

The operators set/cupto rely on the concept of *delimiter* of a continuation, that allows marking the occurrences of set. New delimiters can be created upon request. For two delimiters p_1 and p_2, one can write an expression like the following one

$$1 + \mathsf{set}\ p_1\ \mathsf{in}\ \mathbf{2} + \mathsf{set}\ p_2\ \mathsf{in}\ \mathbf{3} + \mathsf{cupto}\ p_1\ \mathsf{as}\ c\ \mathsf{in}\ (4 \ + \ (c\ 5))$$

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

Application to TDPE. To use set/cupto to address the problem of fixing the relative position of abstractions and discriminators, we must create a new delimiter with each created dynamic $\underline{\lambda}$. Further, we maintain a global list associating to each delimiter a set of variables. To introduce a new δ, we look for all the free variables of its condition, and look in this list for the last delimiter introduced to which one of these variables is associated. Since the term is built in depth first manner and from left to right, one obtains a closed term.

We thus modify the algorithm of TDPE in the following way:

$$\begin{array}{rcl}
\downarrow^{\sigma\to\tau} V & = & \mathsf{let}\ \underline{x}\ \text{be a fresh variable and } p \text{ be a new delimiter} \\
 & & \mathsf{in}\ \underline{\lambda}\,\underline{x}.\,\mathsf{set}\ p\ \mathsf{in} \downarrow^{\tau} (V\ @ \uparrow^{\sigma} \underline{x}) \\
\uparrow^{\sigma_1+\sigma_2} M & = & \mathsf{let}\ m \text{ be the best delimiter for M} \\
 & & \quad \mathsf{in}\ \mathsf{cupto}\ m\ \mathsf{as}\ c \\
 & & \quad\quad \mathsf{in}\ \mathsf{let}\ \underline{x}_1 \text{ and } \underline{x}_2 \text{ be fresh variables,} \\
 & & \quad\quad\quad n_1 = \mathsf{set}\ m\ \mathsf{in}\ (c\ @\ \iota_1(\uparrow^{\sigma_1} \underline{x}_1)), \\
 & & \quad\quad\quad n_2 = \mathsf{set}\ m\ \mathsf{in}\ (c\ @\ \iota_2(\uparrow^{\sigma_2} \underline{x}_2)), \\
 & & \quad\quad\ \mathsf{in}\ \ \underline{\delta}(\ M\ ,\ \underline{x}_1.\ n_1,\ \underline{x}_2.\ n_2)
\end{array}$$

The complete algorithm is presented in Figure 9.

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 [13]). 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.

5 Results and application to isomorphisms of types

Let us observe the code produced by the new normaliser for the function `f` at the end of section 4.1.

```
 # residualise2  (base **-> (base **-> ((base **-> (sum (base ,base))) **->
     ((base **-> (sum (base ,base))))))) f;;
- : normal =
(fun v0 v1 v2 ->
 (match (v2 v1) with
   | Left  v3 -> (fun v5 -> (Left v5))
   | Right  v4 ->
         (match (v2 v0) with
           | Left  v7 -> (fun v6 -> (Left v7))
           | Right  v8 -> (fun v6 -> (Right v8)))
 )
)
```

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

```
 # residualise2 ((bool **-> bool) **-> (bool **-> bool)) fff;;
- : normal =
(fun v0 ->
 (match (v0 (Left ())) with
   | Left  v4 ->
      (match (v0 (Right ())) with
        | Left  v6 -> (fun v1 -> (Left ()))
        | Right  v7 ->
           (fun v1 -> (match v1 with
                       | Left  v2 -> (Left ())
                       | Right  v3 -> (Right ()))
      ))

   | Right  v5 ->
      (match (v0 (Right ())) with
        | Left  v10 ->
           (fun v1 -> (match v1 with
                       | Left  v2 -> (Right ())
                       | Right  v3 -> (Left ()))
```

```
          )
        | Right  v11 -> (fun v1 -> (Right ()))
 )
)
```

This time, the result is identical to the residualisation of the identity:

```
 # residualise2 ((bool **-> bool) **-> (bool **-> bool)) id;;
- : normal =
(fun v0 ->
 (match (v0 (Left ())) with
   | Left  v4 ->
      (match (v0 (Right ())) with
        | Left  v6 -> (fun v1 -> (Left ()))
        | Right  v7 ->
           (fun v1 -> (match v1 with
                       | Left  v2 -> (Left ())
                       | Right  v3 -> (Right ()))
           ))

   | Right  v5 ->
      (match (v0 (Right ())) with
        | Left  v8 ->
           (fun v1 -> (match v1 with
                       | Left  v2 -> (Right ())
                       | Right  v3 -> (Left ()))
           )
        | Right  v9 -> (fun v1 -> (Right ()))
 )
)
```

Figure 10 shows the residualisation of the function `comp3` with the new normaliser. Compared with the result presented at figure 6, it is approximately 48 times smaller (25 lines instead of 1200, and approximately 250 without taking account of the condition (**C**)).

5.1 Eta-reduction

The normal forms produced by the new normaliser are η-long normal forms, as presented in [31]. 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 8).

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 6 give a result of hundreds of lines.

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

```
 # etared (residualise2 ... comp3);;
- : normal = (fun a -> a)
```

6 Conclusions

We presented in [5] an application of the optimization based on the condition (**B**) 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, I mainly took into account the condition (**C**) from [31]. Notice that the modification concerning the constraint (**A**) may entail an increase of the size of the result, because if the δ 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 δ to obtain a canonical form.

As shown in the example of isomorphisms of types, this work has a theoretical interest. More generally, the new normaliser 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 time in the residual program than in the original.

Finally, this work provides a significant test-bed for control operators, since, as we mentioned at the end of section 4.1.3, it provides a realistic application which seems to really make use of the difference in expressivity between shift/reset and cupto/set.

7 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*.

8 Concluding remarks

We have presented a notion of normal term for the typed lambda calculus with sums and proved that every term of the calculus is equivalent to one in normal form. Further, we have used this theoretical development as the basis to implement a partial evaluator that provides a reductionless normalisation procedure for the typed lambda calculus with binary sums.

Our partial evaluator is in the style of TDPE. Thus, it can be grafted on any suitable interpreter, and does not need to examine the structure of the compiled code during normalisation. Its main originality is the use of the control operators set/cupto to fix the relative position of abstractions and discriminators. This is the first non-trivial exploitation of the extra expressive power of set/cupto over shift/reset. The effectiveness of the partial evaluator has been tested on the very sophisticated terms that come from the study of isomorphisms in the typed lambda calculus with sums [6], that make previously existing partial evaluators explode.

The new algorithm does not use all the power of the operators set/cupto. In particular we do not use their ability to code exceptions. One could thus use only a restricted version of these operators. There is, for example, a hierarchical version of shift/reset [13], that allows several, but fixed, levels of control. An implementation with shift/reset (hierarchical or not) is not obvious.

Acknowledgements. Thanks are due to Xavier Leroy for the `call/cc` for *Objective Caml*, and to Olivier Danvy, Andrzej Filinski, and Didier Rémy for interesting discussions about control operators.

References

[1] T. Altenkirch, P. Dybjer, M. Hofmann, and P. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In J. Halpern, editor, *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pages 203–210, Boston, Massachusetts, June 2001. IEEE Computer Society Press.

[2] M.-V. Aponte and R. Di Cosmo. Type isomorphisms for module signatures. In *Programming Languages Implementation and Logic Programming (PLILP)*, volume 1140 of *Lecture Notes in Computer Science*, pages 334–346. Springer-Verlag, 1996.

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

[4] 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 Lecture Notes in Computer Science, pages 240–252, Kyoto, Japan, March 1998. Springer-Verlag.

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

[6] 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.

[7] 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 Lecture Notes in Computer Science, pages 57–71, Genova, Italy, April 2001. Springer-Verlag.

[8] 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*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

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

[10] 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, January 1996. ACM Press.

[11] 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 Lecture Notes in Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag.

[12] O. Danvy and P. Dybjer, editors. *Preliminary Proceedings of the APPSEM Workshop on Normalization by Evaluation*, BRICS Note NS-98-1. Department of Computer Science, University of Aarhus, 1998.

[13] O. Danvy and A. Filinski. Abstracting control. In M. Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.

[14] O. Danvy and A. Filinski. Representing control, a study of the cps transformation. In *Mathematical Structures in Computer Science*, number 2(4), pages 361–191, December 1992.

[15] D. Delahaye, R. Di Cosmo, and B. Werner. Recherche dans une bibliothèque de preuves Coq en utilisant le type et modulo isomorphismes. In *PRC/GDR de programmation, Pôle Preuves et Spécifications Algébriques*, 1997.

[16] R. Di Cosmo. Type isomorphisms in a type assignment framework. In A. W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 200–210, Albuquerque, New Mexico, January 1992. ACM Press.

[17] R. Di Cosmo. Deciding type isomorphisms in a type assignment framework. *Journal of Functional Programming*, 3(3):485–525, 1993.

[18] R. Di Cosmo. *Isomorphisms of types: from λ-calculus to information retrieval and language design*. Birkhauser, 1995. ISBN-0-8176-3763-X.

[19] J. Doner and A. Tarski. An extended arithmetic of ordinal numbers. *Fundamenta Mathematica*, 65:95–127, 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 Press, 2002.

[21] M. Fiore and A. Simpson. Lambda-definability with sums via Grothendieck logical relations. In *Typed Lambda Calculus and Applications*, number 1581 in Lecture Notes in Computer Science. 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):135–141, May 1985.

[25] J. Lambek and P. Scott. *Introduction to higher order categorical logic*, volume 7 of *Cambridge studies in advanced mathematics*. Cambridge University Press, 1986.

[26] M. Rittri. Retrieving library identifiers by equational matching of types. In M. E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, number 449 in Lecture Notes in Computer Science, pages 603–617, Kaiserslautern, Germany, July 1990. Springer-Verlag.

[27] M. Rittri. *Searching program libraries by type and proving compiler correctness by bisimulation*. PhD thesis, University of Göteborg, Göteborg, Sweden, 1990.

[28] M. Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1):71–89, 1991.

[29] C. Runciman and I. Toyn. Retrieving re-usable software components by polymorphic type. *Journal of Functional Programming*, 1(2):191–211, 1991.

[30] S. V. Soloviev. The category of finite sets and cartesian closed categories. *Journal of Soviet Mathematics*, 22(3):1387–1400, 1983.

[31] R. D. C. Vincent Balat and M. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, Venice, Italy, January 2004. ACM Press.

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

```
 # residualise ((bool **-> bool) **-> (bool **-> bool)) fff;;
- : Controls.ans = (fun v0 v1 ->
                    (match v1
                     with
                     | Left
                       v2 -> (match (v0 (Left ()))
                              with
                              | Left  v10 -> (match (v0 (Left ())) with
                                                    | Left  v14 -> (match (v0 (Left ())) with
                                                                          | Left  v16 -> (Left ())
                                                                          | Right  v16 -> (Right ()))

                                                    | Right  v14 -> (match (v0 (Right ())) with
                                                                          | Left  v15 -> (Left ())
                                                                          | Right  v15 -> (Right ()))
                                                      )

                              | Right  v10 -> (match (v0 (Right ())) with
                                                    | Left  v11 -> (match (v0 (Left ())) with
                                                                          | Left  v13 -> (Left ())
                                                                          | Right  v13 -> (Right ()))

                                                    | Right  v11 -> (match (v0 (Right ())) with
                                                                          | Left  v12 -> (Left ())
                                                                          | Right  v12 -> (Right ()))
                                                      )
                                )

                     | Right
                       v2 -> (match (v0 (Right ()))
                              with
                              | Left  v3 -> (match (v0 (Left ())) with
                                                   | Left  v7 -> (match (v0 (Left ())) with
                                                                        | Left  v9 -> (Left ())
                                                                        | Right  v9 -> (Right ()))

                                                   | Right  v7 -> (match (v0 (Right ())) with
                                                                        | Left  v8 -> (Left ())
                                                                        | Right  v8 -> (Right ()))
                                                     )

                              | Right  v3 -> (match (v0 (Right ())) with
                                                   | Left  v4 -> (match (v0 (Left ())) with
                                                                        | Left  v6 -> (Left ())
                                                                        | Right  v6 -> (Right ()))

                                                   | Right  v4 -> (match (v0 (Right ())) with
                                                                        | Left  v5 -> (Left ())
                                                                        | Right  v5 -> (Right ()))
                                                     )
                                )))
```

Figure 4: Normalisation of `fff` by TDPE.

```
let f3 =
  fun (a1,a2) -> (
    (fun u ->
      match (a2 u) with
        Left b -> Left (fun v ->
          match (a1 v) with
            Left c -> (c u)
          | Right c -> (match (c u),(b v) with
              (Left d), (Left e) -> (* y2,y3 *) Left (fst d)
            | (Left d), (Right e) -> (* y2,x3 *) Right (fst e)
            | (Right (Left d)), (Left e) -> (* yx,y3 *) Right (snd d)
            | (Right (Left d)), (Right e) -> (* yx,x3 *) Left (fst d)
            | (Right (Right d)), (Left e) -> (* x2,y3 *) Left (fst e)
            | (Right (Right d)), (Right e) -> (* x2,x3 *) Right (fst d))
                    )
      | Right b -> Right (fun v ->
          match (a1 v) with
          | Left c -> (match (c u),(b v) with
              (Left d), (Left e) -> (* y,y4 *) Left (d,(fst e))
            | (Left d), (Right (Left e)) -> (* y,y2x2 *) Right (Right (snd (snd e)))
            | (Left d), (Right (Right e)) -> (* y,x4 *) Right (Left ((d,fst e)))
            | (Right d), (Left e) -> (* x,y4 *) Right (Left ((fst e,d)))
            | (Right d), (Right (Left e)) -> (* x,y2x2 *) Left (fst e,(fst (snd e)))
            | (Right d), (Right (Right e)) -> (* x,x4 *) Right (Right (d,fst e)))
          | Right c -> (c u)
    )),
    (fun v ->
      match (a1 v) with
        Left c -> Left (fun u ->
          match (a2 u) with
            Left b -> (b v)
          | Right b -> (match (c u),(b v) with
              (Left d), (Left e) -> (* y,y4 *) Left (snd e)
            | (Left d), (Right (Left e)) -> (* y,y2x2 *) Left (d,((fst e),(fst (snd e))))
            | (Left d), (Right (Right e)) -> (* y,x4 *) Right (snd e)
            | (Right d), (Left e) -> (* x,y4 *) Left (snd e)
            | (Right d), (Right (Left e)) -> (* x,y2x2 *) Right (d,(snd (snd e)))
            | (Right d), (Right (Right e)) -> (* x,x4 *) Right (snd e))
                    )
      | Right c -> Right (fun u ->
          match (a2 u) with
          | Left b -> (match (c u),(b v) with
              (Left d), (Left e) -> (* y2,y3 *) Left ((snd d),e)
            | (Left d), (Right e) -> (* y2,x3 *) Right (Left (fst d,(snd d,snd e)))
            | (Right (Left d)), (Left e) -> (* yx,y3 *) Left ((fst d),e)
            | (Right (Left d)), (Right e) -> (* yx,x3 *) Right (Right ((snd d),e))
            | (Right (Right d)), (Left e) -> (* x2,y3 *) Right (Left ((fst (snd e)),
                                                                      ((snd (snd e)),d)))
            | (Right (Right d)), (Right e) -> (* x2,x3 *) Right (Right ((snd d),e)))
          | Right b -> (b v)
   )));;

val f3 : ('a -> ('b -> ('c, 'd) sum, 'b -> ('c * 'c, ('c * 'd, 'd * 'd) sum) sum) sum) *
('b -> ('a -> ('c * ('c * 'c), 'd * ('d * 'd)) sum,
'a -> ('c * ('c * ('c * 'c)), ('c * ('c * ('d * 'd)),
'd * ('d * ('d * 'd))) sum) sum) sum)
->
('b -> ('a -> ('c, 'd) sum, 'a -> ('c * 'c, ('c * 'd, 'd * 'd) sum) sum) sum) *
('a -> ('b -> ('c * ('c * 'c), 'd * ('d * 'd)) sum,
'b -> ('c * ('c * ('c * 'c)), ('c * ('c * ('d * 'd)),
'd * ('d * ('d * 'd))) sum) sum) sum) = <fun>
```

Figure 5: Isomorphism in the case $n = 3$.

```
 #  let comp3 x = f3 (f3 x);;

 #  residualise ... comp3;;

- : Controls.ans =
(fun a ->
 ((fun u ->
   (match ((proj1 a) u)
    with
    | Left v160 ->
        (Left (fun v ->
           (match ((proj2 a) v)
            with
            | Left v241 -> (match ((proj1 a) u)
                with
                | Left  v313 -> (match (v313 v)
                   with
                   | Left  v319 -> (Left v319)
                   | Right  v319 -> (Right v319))

                | Right  v313 -> (match (v313 v)
                   with
                   | Left  v314 -> (match (v241 u) with
                                       | Left  v318 -> (Left (proj1 v314))
                                       | Right  v318 -> (Right (proj1 v318)))

                   | Right  v314 -> (match v314 with
                                       | Left  v315 -> (match (v241 u) with
                                                           | Left  v317 -> (Right (proj2 v315))
                                                           | Right  v317 -> (Left (proj1 v315)))

                                       | Right  v315 -> (match (v241 u) with
                                                           | Left  v316 -> (Left (proj1 v316))
                                                           | Right  v316 -> (Right (proj1 v315)))
                                    )
                                 )
                              )

            | Right v241 -> (match ((proj1 a) u)
                with
                | Left v242 -> (match (v242 v) with
                                  ...
...
```

Figure 6: Residualisation of the composition of `f3` with itself (small extract).

$$\frac{}{\Gamma, x : \tau, \Gamma' \vdash_{\mathcal{M}_0} x : \tau}$$

$$\frac{\Gamma \vdash_{\mathcal{M}_0} M : \tau_1 \times \tau_2}{\Gamma \vdash_{\mathcal{M}_0} \pi_i(M) : \tau_i} \ (i = 1, 2)$$

$$\frac{\Gamma \vdash_{\mathcal{M}_0} M : \tau_1 \to \tau \qquad \Gamma \vdash_{\mathcal{N}_0} N : \tau_1}{\Gamma \vdash_{\mathcal{M}_0} M \mathbin{@} N : \tau}$$

$$\frac{\Gamma \vdash_{\mathcal{M}_0} M : \tau}{\Gamma \vdash_{\mathcal{M}} M : \tau}$$

$$\frac{\Gamma \vdash_{\mathcal{M}_0} M : \tau_1 + \tau_2 \qquad \Gamma,\ x_i : \tau_i \vdash_{\mathcal{M}} M_i : \tau \ (i = 1, 2)}{\Gamma \vdash_{\mathcal{M}} \delta(M, x_1.\ M_1, x_2.\ M_2) : \tau}$$

$$\frac{\Gamma \vdash_{\mathcal{M}_0} M : \theta}{\Gamma \vdash_{\mathcal{N}_0} M : \theta} \ (\theta \text{ a base type})$$

$$\frac{}{\Gamma \vdash_{\mathcal{N}_0} \langle\rangle : 1} \qquad \frac{\Gamma \vdash_{\mathcal{N}_0} N_i : \tau_i \ (i = 1, 2)}{\Gamma \vdash_{\mathcal{N}_0} \mathrm{pair}\langle N_1, N_2\rangle : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash_{\mathcal{N}_0} N : \tau_i}{\Gamma \vdash_{\mathcal{N}_0} \iota_i^{\tau_1,\tau_2}(N) : \tau_1 + \tau_2} \ (i = 1, 2)$$

$$\frac{\Gamma \vdash_{\mathcal{N}_0} N : \tau}{\Gamma \vdash_{\mathcal{N}} N : \tau}$$

$$\frac{\Gamma,\ x : \tau_1 \vdash_{\mathcal{N}} N : \tau}{\Gamma \vdash_{\mathcal{N}_0} \lambda x : \tau_1.\ N : \tau_1 \to \tau} \ \left(\ x \in \mathrm{FV}(C) \text{ for all } C \in \mathrm{Guards}(N) \ \right)$$

$$\frac{\Gamma \vdash_{\mathcal{M}_0} M : \tau_1 + \tau_2 \qquad \Gamma,\ x_i : \tau_i \vdash_{\mathcal{N}} N_i : \tau \ (i = 1, 2)}{\Gamma \vdash_{\mathcal{N}} \delta(M, x_1.\ N_1, x_2.\ N_2) : \tau} \ \left(\begin{array}{l} M \not\approx C \text{ for all } C \in \bigcup_{i=1,2} \mathrm{Guards}(x_i.\ N_i) \\ N_1 \not\approx N_2 \text{ whenever } x_1 \notin \mathrm{FV}(N_1) \text{ and } x_2 \notin \mathrm{FV}(N_2) \end{array} \right)$$

$$\mathrm{Guards}(N) \stackrel{\mathrm{def}}{=} \begin{cases} \{ M \} \cup \bigcup_{i=1,2} \mathrm{Guards}(x_i.\ N_i) & \text{, if } N = \delta(M, x_1.\ N_1, x_2.\ N_2) \\ \emptyset & \text{, otherwise} \end{cases}$$

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

Figure 7: Neutral and normal terms.
(The context is assumed consistent unless stated otherwise.)

$$
\begin{array}{rcl}
\delta\,(\delta\,(t,\ \iota_1 \circ h_1,\ \iota_2 \circ h_2),\ f,\ g) & =_{\beta\eta} & \delta\,(t,\ f \circ h_1,\ g \circ h_2) \\
(f((\delta(t,\ x_1.\ t_1,\ x_2.\ t_2)))) & =_{\beta\eta} & \delta(t,\ x_1.\ (f(t_1)),\ x_2.\ (f(t_2))) \\
((\delta(t,\ x_1.\ t_1,\ x_2.\ t_2))(t')) & =_{\beta\eta} & \delta(t,\ x_1.\ (t_1(t')),\ x_2.\ (t_2(t'))) \\
\delta(t,\ x.\ \delta(t,\ x_1.\ t_1,\ x_2.\ t_2),\ y.\ u) & =_{\beta\eta} & \delta(t,\ x.\ t_1[x/x_1],\ y.\ u) \\
\delta(t,\ x.\ \delta(t',\ x_1.\ u_1,\ x_2.\ u_2),\ y.\ u) & =_{\beta\eta} & \delta(t',\ x_1.\ \delta(t,\ x.\ u_1,\ y.\ u),\ x_2.\ \delta(t,\ x.\ u_2,\ y.\ u))
\end{array}
$$

if $x \notin \mathrm{FV}(t')$ and $x_i \notin \mathrm{FV}(t)$ $(i = 1, 2)$

Figure 8: Some examples of $\beta\eta$-conversion with sum types

$$
\begin{array}{rcl}
\downarrow^{\theta} V & = & V \\
\downarrow^{1} V & = & \underline{\langle\rangle} \\
\downarrow^{\sigma\to\tau} V & = & \text{let } \underline{x} \text{ be a fresh variable and } p \text{ a new delimiter in } \underline{\lambda}\,\underline{x}.\ \text{set } p \text{ in} \downarrow^{\tau} (V\ @\ \uparrow^{\sigma} \underline{x}) \\
\downarrow^{\tau_1\times\tau_2} V & = & \underline{\mathrm{pair}}\langle\downarrow^{\tau_1} (\pi_1(V)),\ \downarrow^{\tau_2} (\pi_2(V))\rangle \\
\downarrow^{\tau_1+\tau_2} V & = & \delta\left(V,\ x_1.\ \underline{\iota_1}(\downarrow^{\tau_1} x_1),\ x_2.\ \underline{\iota_2}(\downarrow^{\tau_2} x_2)\right) \\
\uparrow^{\theta} M & = & M \\
\uparrow^{1} M & = & \langle\rangle \\
\uparrow^{\tau\to\sigma} M & = & \lambda\, x.\uparrow^{\sigma} (M\ \underline{@}\downarrow^{\tau} x) \\
\uparrow^{\sigma_1\times\sigma_2} M & = & \mathrm{pair}\langle\uparrow^{\sigma_1} (\underline{\pi_1}(M)),\ \uparrow^{\sigma_2} (\underline{\pi_2}(M))\rangle \\
\uparrow^{\sigma_1+\sigma_2} M & = & \text{if } M \text{ is globally associated to (L, } \underline{z}\text{) modulo} \approx \\
 & & \text{then } \iota_1(\uparrow^{\sigma_1} \underline{z}) \\
 & & \text{else if } M \text{ is globally associated to (R, } \underline{z}\text{) modulo} \approx \\
 & & \quad \text{then } \iota_2(\uparrow^{\sigma_2} \underline{z}) \\
 & & \quad \text{else let } m \text{ be the best delimiter for M} \\
 & & \quad\quad \text{in cupto } m \text{ as } c \\
 & & \quad\quad\quad \text{in let } \underline{x}_1 \text{ and } \underline{x}_2 \text{ be fresh variables} \\
 & & \quad\quad\quad\quad \text{associate } M \text{ to } (L,\ \underline{x}_1) \text{ while computing } n_1 = \text{set } m \text{ in } (c\ @\ \iota_1(\uparrow^{\sigma_1} \underline{x}_1)) \\
 & & \quad\quad\quad\quad \text{associate } M \text{ to } (R,\ \underline{x}_2) \text{ while computing } n_2 = \text{set } m \text{ in } (c\ @\ \iota_2(\uparrow^{\sigma_2} \underline{x}_2)) \\
 & & \quad\quad\quad\quad\quad \text{in if } x_1 \notin \mathrm{FV}(n_1), x_2 \notin \mathrm{FV}(n_2), \text{and } n_1 \approx n_2 \\
 & & \quad\quad\quad\quad\quad\quad \text{then } n_1 \\
 & & \quad\quad\quad\quad\quad\quad \text{else } \underline{\delta}\,(M,\ \underline{x}_1.\ n_1,\ \underline{x}_2.\ n_2)
\end{array}
$$

Figure 9: Optimised type-directed normalisation.

```
 #  residualise2 ... comp3;;
- : normal =
(fun a ->
 ((fun u -> (match ((proj1 a) u) with
    | Left  v32 -> (Left (fun v -> (match (v32 v) with
         | Left  v36 -> (Left v36)
         | Right  v37 -> (Right v37))
                    ))
    | Right  v33 -> (Right (fun v -> (match (v33 v) with
         | Left  v50 -> (Left ((proj1 v50) , (proj2 v50)))
         | Right  v51 -> (match v51 with
                           | Left  v54 -> (Right (Left ((proj1 v54) , (proj2 v54))))
                           | Right  v55 -> (Right (Right ((proj1 v55) , (proj2 v55)))))
          ))))
  ) ,
  (fun v ->
   (match ((proj2 a) v) with
    | Left  v0 -> (Left (fun u -> (match (v0 u) with
        | Left  v4 -> (Left ((proj1 v4) , ((proj1 (proj2 v4)) , (proj2 (proj2 v4)))))
        | Right  v5 -> (Right ((proj1 v5) , ((proj1 (proj2 v5)) , (proj2 (proj2 v5))))))
                    ))
    | Right  v1 -> (Right (fun u -> (match (v1 u) with
        | Left  v20 -> (Left ((proj1 v20) , ((proj1 (proj2 v20)) ,
                                  ((proj1 (proj2 (proj2 v20))) , (proj2 (proj2 (proj2 v20)))))))
        | Right  v21 -> (match v21 with
                          | Left  v22 -> (Right (Left ((proj1 v22) , ((proj1 (proj2 v22)),
                                    ((proj1 (proj2 (proj2 v22))) , (proj2 (proj2 (proj2 v22))))))))
                          | Right  v23 -> (Right (Right ((proj1 v23) , ((proj1 (proj2 v23)) ,
                                    ((proj1 (proj2 (proj2 v23))) , (proj2 (proj2 (proj2 v23)))))))))
         )))))))
```

Figure 10: Residualisation of the composition of `f3` with itself, with the new algorithm.