Memoization in Type-Directed Partial Evaluation

Vincent Balat[1] and Olivier Danvy[2]

[1] PPS, Université Paris VII – Denis Diderot
Case 7014, 2 place Jussieu, F-75251 Paris Cedex 05, France
(balat@pps.jussieu.fr)

[2] BRICS*
Department of Computer Science, University of Aarhus
Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark
(danvy@brics.dk)

Abstract. We use a code generator—type-directed partial evaluation—to verify conversions between isomorphic types, or more precisely to verify that a composite function is the identity function at some complicated type. A typed functional language such as ML provides a natural support to express the functions and type-directed partial evaluation provides a convenient setting to obtain the normal form of their composition. However, off-the-shelf type-directed partial evaluation turns out to yield gigantic normal forms.
We identify that this gigantism is due to redundancies, and that these redundancies originate in the handling of sums, which uses delimited continuations. We successfully eliminate these redundancies by extending type-directed partial evaluation with memoization capabilities. The result only works for pure functional programs, but it provides an unexpected use of code generation and it yields orders-of-magnitude improvements both in time and in space for type isomorphisms.

1 Introduction

1.1 Background: reduction-based vs. reduction-free normalization

Say that we consider binary trees modulo associativity. Binary trees are easily coded as a data type in Standard ML [36, 37, 45]:

```
datatype 'a bt = LEAF of 'a
               | NODE of 'a bt * 'a bt
```

(In this declaration, `bt` is the name of the data type. It is parameterized with a type variable `'a` to express that the data type is polymorphic: we can represent

* Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

trees of integers, trees of reals, trees of lists, etc. `LEAF` and `NODE` are binary-tree constructors.)

The following conversion rule (written with an infix $\Leftrightarrow$) defines associativity:

```
∀ t1, t2, t3 : 'a bt, NODE (t1, NODE (t2, t3)) ⇔ NODE (NODE (t1, t2), t3).
```

Two binary trees are equal modulo associativity if they can be converted to each other using $\Leftrightarrow$.

How do we represent binary trees modulo associativity in practice? One option is to use the data type above and test for equality using the conversion rule. A more efficient version, however, exists. It is based on the idea of *orienting* the conversion rule into a rewriting rule. For example, we could orient is as follows:

```
∀ t1, t2, t3 : 'a bt, NODE (t1, NODE (t2, t3)) ⇐ NODE (NODE (t1, t2), t3).
```

This rewriting rule is nice because repeatedly applying it (1) terminates and (2) yields a unique normal form. (A normal form here is a binary tree for which the rewriting rule cannot be applied.) Representing a binary tree modulo associativity is thus best done with its normal form because it is more efficient to test for equality.

In the present case, the data type of binary trees in normal form can be coded as follows:

```
datatype 'a bt_nf = LEAF_nf of 'a
                  | NODE_nf of 'a * 'a bt_nf
```

The constructor `NODE_nf` guarantees that the rewriting rule cannot be applied.

Since this data type is isomorphic to the data type of non-empty lists, we can represent normalized binary trees as ML lists. The question then is how to normalize binary trees. In ML terms, this amounts to writing a function

```
normalize : 'a bt -> 'a list
```

that flattens its argument into a normal form.

The traditional, reduction-based, approach is to traverse the source tree and repeatedly apply the rewrite rule:

```
fun reduction_based_normalize (LEAF x)
    = x :: nil
  | reduction_based_normalize (NODE (LEAF x, t))
    = x :: (reduction_based_normalize t)
  | reduction_based_normalize (NODE (NODE (t1, t2), t3))
    = reduction_based_normalize (NODE (t1, NODE (t2, t3)))
```

An alternative, reduction-free, approach to normalization, however, exists: it amounts to interpreting the binary tree in a non-standard model and inverting this interpretation. In the present case, we choose the non-standard model to be the function space

```
'a list -> 'a list
```

We map leaves into a function that adds an element to its argument, we map nodes into function composition, and we invert the interpretation function by applying values to the empty list:

```
fun reduction_free_normalize t
    = let fun eval (LEAF x)
              = (fn a => x :: a)
            | eval (NODE (t1, t2))
              = (eval t1) o (eval t2)
          fun reify value
              = value nil
      in reify (eval t)
      end
```

This seemingly daunting function can be simplified as follows: rather than returning a function, the argument of this function can be specified as one more argument to `eval`, and `reify` can be inlined:

```
fun reduction_free_normalize_simpler t
    = let fun eval (LEAF x) a
              = x :: a
            | eval (NODE (t1, t2)) a
              = eval t1 (eval t2 a)
      in eval t nil
      end
```

The result is the familiar flatten function with an accumulator.

This way of normalizing binary trees is said to be reduction free because it does not explicitly apply the rewriting rule. Because it normalizes a term by inverting an evaluation function (into a non-standard model), reduction-free normalization is also referred to as *normalization by evaluation*. The flattening example above is folklore in the normalization-by-evaluation community.

Normalization by evaluation has been variously studied in logic, proof theory, and category theory [2, 3, 8–10, 13] and in partial evaluation [14, 16]. Type-directed partial evaluation, which we present next, has been investigated both practically [5, 15, 17, 18, 29, 31, 38] and foundationally [24, 25, 47].

1.2 Type-directed partial evaluation

Type-directed partial evaluation is a practical instance of normalization by evaluation and is used for specializing functional programs. The evaluation function it inverts is the standard evaluation of functional programs. Consequently, a type-directed partial evaluator maps *values* to a textual representation of their normal form, in contrast to a traditional syntax-directed partial evaluator, which maps the *textual representation* of a source program to the textual representation of the corresponding specialized program.

In the present work, we consider a pure version of type-directed partial evaluation for ML with the following types (a is atomic):

$$t ::= a \mid t_1 \to t_2 \mid t_1 \times t_2 \mid t_1 + t_2$$

For example, let us consider the following ML function, which exponentiates its argument `x` by recursively halving its argument `n`, using the auxiliary function `binary`. Depending on the parity of its argument, `binary` applies `odd` or `even` to each intermediate result. The functions `quot` and `rem` respectively compute the quotient and the remainder of two integers; they are found in the `Int` library. The opportunity for specialization here is that the first argument of `exponentiate` (and thus the argument of `binary`) is known statically.

```
fun exponentiate n (odd, even) x
    = let fun binary 0
              = x
            | binary n
              = let val r = binary (Int.quot (n, 2))
                in if Int.rem (n, 2) = 0
                   then even r
                   else odd r
                end
      in binary n
      end
```

A syntax-directed partial evaluator maps the textual representation of `exponentiate 20` to the textual representation of its specialized version (the overhead of the interpretation of `20` has been completely eliminated):

```
fn (p1, p2) => fn x3 => let val r4 = p1 x3
                            val r5 = p2 r4
                            val r6 = p1 r5
                            val r7 = p2 r6
                        in p2 r7
                        end
```

In contrast, a type-directed partial evaluator maps the value of `exponentiate 20` (together with a representation of its type) to the textual representation of its specialized version. (In fact, the residual function above is the actual output of our type-directed partial evaluator.)

1.3 Motivation

Recently, we have realized that a proof-theoretical application of type-directed partial evaluation was affected by the size and redundancy of the generated code. Unsurprisingly, we have diagnosed the problem to arise because of sums, which are handled with continuations and therefore duplicate contexts.

For example, at type $(a \to b) \to (c \to a) \to c \to b$, the term `fn f => fn g => fn x => f (g x)` is normalized into the following residual term:

```
fn x0 => fn x1 => fn x2 => let val r3 = x1 x2
                           in x0 r3
                           end
```

At type $(a \to b) \to (bool \to a) \to bool \to b$, however, it is normalized into the following other residual term, where the application of `x0` occurs in both conditional branches:[1]

```
fn x0 => fn x1 => fn x2 => if x2
                           then let val r3 = x1 true
                                in x0 r3
                                end
                           else let val r5 = x1 false
                                in x0 r5
                                end
```

In both cases, the residual term is in normal form: it contains no function applications or conditional expressions that could be simplified away. It is also fully eta-expanded.

Normalization at boolean type is handled by duplicating contexts (the application of `x0` in the example just above). This duplication is known to yield redundant residual terms in a pure setting. For example, normalizing `fn f => fn g => fn x => f (g x) (g x)` at type $(bool \to bool \to a) \to (b \to bool) \to b \to a$ yields the following residual term:

```
fn x0 => fn x1 => fn x2 => let val r3 = x1 x2
                           in if r3
                              then let val r4 = x0 true
                                       val r5 = x1 x2
                                   in if r5
                                      then r4 true
                                      else r4 false
                                   end
                              else let val r8 = x0 false
                                       val r9 = x1 x2
                                   in if r9
                                      then r8 true
                                      else r8 false
                                   end
                           end
```

This residual term is redundant in a pure setting because in each branch of the outer conditional expression, we know the result of `x1 x2` and therefore there is no need to recompute it and test it. The residual term could thus be simplified into the following one:

```
fn x0 => fn x1 => fn x2 => let val r3 = x1 x2
                           in if r3
                              then let val r4 = x0 true
                                   in r4 true
                                   end
```

[1] The boolean type is a trivial sum.

```
                    else let val r6 = x0 false
                         in r6 false
                         end
            end
```

In the proof-theoretic setting considered here (see Section 2), such a simplification is crucial.

1.4 Contribution and overview

We solve the above redundancy by introducing a memoization mechanism in type-directed partial evaluation.

The rest of this article is organized as follows: Section 2 describes the proof-theoretical setting of our work; Section 3 reviews type-directed partial evaluation; and Section 4 presents the memoization mechanism.

2 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 [20–22, 39–42] and to match modules by specifications [7, 19].

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 recent joint work by Balat, Di Cosmo, and 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" [23].

2.1 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 [46]. Furthermore, Gurevič has shown that in that case, equalities are not finitely axiomatizable [30]. 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.

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

If one replaces sums, 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 [11, 43].

Continuing the parallel with arithmetic, Balat, Di Cosmo, and Fiore have studied the case of isomorphisms of types with empty and sum types [6]. They have 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
$$\begin{aligned} A &= y + x \\ B_n &= y^{n-1} + xy^{n-2} + x^2y^{n-3} + \ldots + x^{n-2}y + x^{n-1} \\ &= \textstyle\sum_{i=0}^{n-1} x^i y^{n-i-1} \\ C_n &= y^n + x^n \\ D_n &= y^{2n-2} + x^2y^{2n-4} + x^4y^{2n-6} \ldots + x^{2n-4}y^2 + x^{2n-2} \\ &= \textstyle\sum_{i=0}^{n-1} x^{2i} y^{2n-2i-2} \end{aligned}$$

Balat, Di Cosmo, and Fiore have proven that these equalities hold in the world of type isomorphisms as well. They did so by exhibiting a family of functions and their inverses. Figure 1 shows a fragment of one of these functions, written in Standard ML, when $n = 3$. The type of this term fragment is displayed at the top of the figure. It corresponds to $(A^u + B_3^u)^v \cdot ... \to (A^v + B_3^v)^u \cdot ...$, 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).[2]

For such large and interlaced functions, whether intuited or automatically produced, 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], chiefly because of commuting conversions [27]. Type-directed partial evaluation does handle sums, but the redundancy pointed out in Section 1.3 is a major impediment.

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

```
('a -> ('b -> ('c,'d) sum, 'b -> ('d * 'd,('d * 'c,'c * 'c) sum) sum) sum) * ...
->
('b -> ('a -> ('c,'d) sum, 'a -> ('d * 'd,('d * 'c,'c * 'c) sum) sum) sum) * ...
```

```
fn (p1, p2)
=> (fn x3
    => (case p2 x3
          of (LEFT s5)
             => LEFT (fn x7
                      => (case p1 x7
                            of (LEFT s9)
                               => (case s9 x3
                                     of (LEFT s12)
                                        => LEFT s12
                                      | (RIGHT s13)
                                        => RIGHT s13)
                             | (RIGHT s10)
                               => (case s5 x7
                                     of (LEFT (p17, (p19, p20)))
                                        => (case s10 x3
                                              of (LEFT (p24, p25))
                                                 => RIGHT p25
                                               | (RIGHT s23)
                                                 => (case s23
                                                       of (LEFT (p28, p29))
                                                          => LEFT p29
                                                        | (RIGHT (p30, p31))
                                                          => RIGHT p20))
                                      | (RIGHT (p32, (p34, p35)))
                                        => (case s10 x3
                                              of (LEFT (p39, p40))
                                                 => LEFT p35
                                               | (RIGHT s38)
                                                 => (case s38
                                                       of (LEFT (p43, p44))
                                                          => RIGHT p43
                                                        | (RIGHT (p45, p46))
                                                          => LEFT p46)))))
           | (RIGHT s6) => RIGHT (fn x47 => ...)),
  fn x111
  => (case p1 x111
        of (LEFT s113) => LEFT (fn x115 => ...)
         | (RIGHT s114) => RIGHT (fn x179 => ...)))
```

Fig. 1. Isomorphism function for $n = 3$ (fragment)

3 Type-directed partial evaluation

Type-directed partial evaluation is defined as a pair of functions for each type constructor. The first function, reify, maps a value into a representation of its normal form. The second function, reflect, maps the representation of a normal form into the corresponding value. Reification and reflection are already well described in the literature [8, 10, 16, 25, 28, 29, 31, 38, 47] and therefore, rather than repeating these descriptions, let us instead focus on the one equation of interest: reflection at sum type.

$$\uparrow_{t_1+t_2} e = \text{shift } \kappa \text{ in } \underline{\text{case}}(e,\ x.\text{reset}(\kappa(\text{in}_1(\uparrow_{t_1} x))),\ y.\text{reset}(\kappa(\text{in}_2(\uparrow_{t_2} y))))$$
$$\text{where } x \text{ and } y \text{ are fresh}$$

The control operator shift abstracts the evaluation context of $\uparrow_{t_1+t_2} e$ and relocates it in each branch of a residual conditional expression. The control operator reset delimits the extent of any subsequent control abstraction in the conditional branches. The effect of this context duplication has been illustrated in Section 1.3.

4 Memoization

4.1 What

Our aim is to avoid dead branches in the residual code by integrating the two following transformations in our type-directed partial evaluator:

$$\begin{aligned}&\text{case}(e,\ x_1.M_1\left[\text{case}(e,\ y_1.N_1,\ y_2.N_2)/_z\right],\ x_2.M_2)\\ &\longrightarrow \text{case}(e,\ x_1.M_1\left[N_1\left[x_1/_{y_1}\right]/_z\right],\ x_2.M_2)\end{aligned}$$

$$\begin{aligned}&\text{case}(e,\ x_1.M_1,\ x_2.M_2\left[\text{case}(e,\ y_1.N_1,\ y_2.N_2)/_z\right])\\ &\longrightarrow \text{case}(e,\ x_1.M_1,\ x_2.M_2\left[N_2\left[x_2/_{y_2}\right]/_z\right])\end{aligned}$$

These transformations are easily derivable from the η rule for sum types:

$$\text{case}\,(t,\ x_1.h\ (\text{in}_1\ x_1),\ x_2.h\ (\text{in}_2\ x_2)) = h\ t$$

For example, taking $h = \lambda x.\ \text{case}(x,\ x_1.M_1\left[\text{case}(x,\ y_1.N_1,\ y_2.N_2)/_z\right],\ x_2.M_2)$ and β-reducing yields the first transformation.

4.2 How

The residual program is an abstract-syntax tree. This abstract-syntax tree is constructed depth first, left to right. Our key idea is to maintain a global stack accounting for conditional branches in the path from the root of the residual program to the current point of construction.

The global stack can be implemented with a global reference and sequential push and pop operations as the construction proceeds. It seems plausible that the correctness of this state-based version of type-directed partial evaluation can be approached by adding a state monad to Filinski's formalization [25]. We are currently looking into this issue [4].

The stack associates a flag (Left or Right) and a variable to an expression as specified below:

$$\uparrow_{t_1+t_2} e = \begin{cases} \mathrm{in}_1(\uparrow_{t_1} z) & \text{if } e \text{ is globally associated to (Left, } z) \\ \mathrm{in}_2(\uparrow_{t_2} z) & \text{if } e \text{ is globally associated to (Right, } z) \\ \mathrm{shift}\ \kappa\ \mathrm{in}\ \underline{\mathrm{case}}(e, & \text{otherwise} \\ \qquad x.\mathrm{reset}(\kappa(\mathrm{in}_1(\uparrow_{t_1} x))), & \\ \qquad y.\mathrm{reset}(\kappa(\mathrm{in}_2(\uparrow_{t_2} y)))) & \\ \text{where } x \text{ and } y \text{ are fresh} & \end{cases}$$

If e is not associated to anything in the stack, then we associate it to (Left, x) when processing the consequent and to (Right, x) when processing the alternative.

4.3 Application

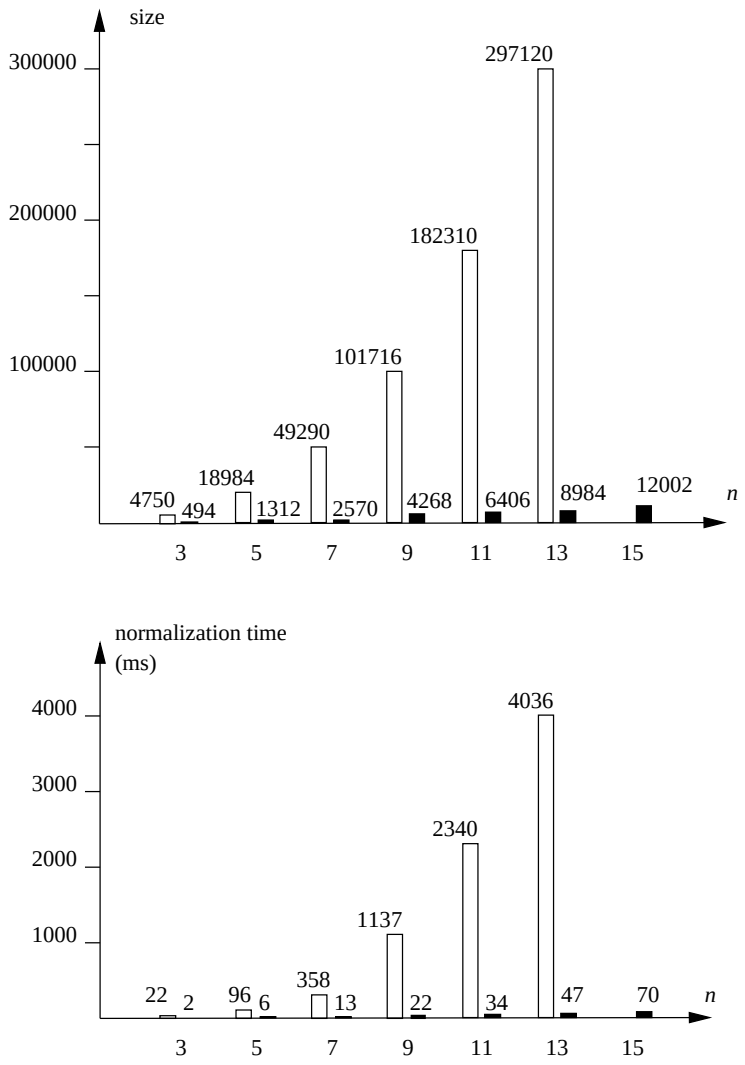
In the present case, memoization pays off: as illustrated in Figure 2, the output of type-directed partial evaluation is between one and two orders of magnitude smaller, for a residualization time that is also between one and two orders of magnitude smaller. (We also observed that the time ML takes for inferring the types of the isomorphism functions offsets the time taken by type-directed partial evaluation, even in the absence of memoization.)

4.4 Common sub-expression elimination

Furthermore, we are now in position to integrate common sub-expression elimination by reflecting at function type into memo functions [32, 35]. These memo functions are indexed by the global stack to ensure their coherence, since a reflected function can be applied in conditional branches with distinct lexical scope. (In the absence of sums, the memo functions can be used uniformly.)

To illustrate common sub-expression elimination, let us come back to the last example of Section 1.3, `fn f => fn g => fn x => f (g x) (g x)`. Without memo functions, residualizing it at type $(a \to a \to b) \to (c \to a) \to c \to b$ yields the following residual term where the application `x1 x2` occurs twice:

```
fn x0 => fn x1 => fn x2 => let val r3 = x1 x2
                               val r4 = x0 r3
                               val r5 = x1 x2
                           in r4 r5
                           end
```



The two graphs visualize the size of the residual abstract syntax trees (the number of their nodes) and their normalization time (in milliseconds, on a 4-processor Sparc station running SunOS 5.7 using SML/NJ Version 110.0.6) for the isomorphism functions described in Section 2 and Figure 1, for $n = 3$, 5, 7, 9, 11, and 13. The white bars account for standard type-directed partial evaluation, and the black bars account for type-directed partial evaluation with memoization.

Fig. 2. Benchmarks

In contrast, memo functions make it possible to obtain the following residual term, where the result `r3` is used twice and thus the application `x1 x2` occurs only once:

```
fn x0 => fn x1 => fn x2 => let val r3 = x1 x2
                               val r4 = x0 r3
                           in r4 r3
                           end
```

5 Related work

Memoisation is a standard component of polyvariant partial evaluators that yield mutually recursive residual programs [12, 33, 44]. Using a traditional syntax-directed partial evaluator, however, is not a realistic option here because our source programs are higher-order and thus require a frightful number of binding-time improvements.

We are not aware of any similar work on type isomorphisms.

Finally, and as illustrated in Section 1.1, type-directed partial evaluation is only one instance of normalization by evaluation. We are not aware of any other use of memoization in other instances.

6 Conclusion and issues

We have extended type-directed partial evaluation of pure functional programs with memoization capabilities. Primarily, memoization makes it possible to keep track of the dynamic result of tests in conditional branches, as in Futamura's Generalized Partial Computation [26]. Secondarily, memoization makes it possible to integrate a form of common sub-expression elimination in type-directed partial evaluation. Getting back to our initial motivation, memoization makes it practical to use type-directed partial evaluation to verify type isomorphisms in the presence of sums.

Acknowledgments: This article sprang from the two authors's participation to the 30th spring school on theoretical computer science (`http://www.pps.jussieu.fr/~ecole`) held in Agay, France, in March 2002, and it has benefited from discussions with Andrzej Filinski and Samuel Lindley. Thanks are also due to Mads Sig Ager, Peter Thiemann, and the reviewers for helpful comments, and to Henning Korsholm Rohde for automating the generation of the isomorphism functions.

This work is supported by the ESPRIT Working Group APPSEM (`http://www.md.chalmers.se/Cs/Research/Semantics/APPSEM`).

References

1. Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip Scott. Normalization by evaluation for typed lambda calculus with coproducts. In Joseph 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. Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction-free normalization proof. In David H. Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Category Theory and Computer Science*, number 953 in Lecture Notes in Computer Science, pages 182–199, Cambridge, UK, August 1995. Springer-Verlag.
3. Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Reduction-free normalisation for a polymorphic system. In Edmund M. Clarke, editor, *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 98–106, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.
4. Vincent Balat. PhD thesis, PPS, Université Paris VII – Denis Diderot, Paris, France, 2002. Forthcoming.
5. Vincent Balat and Olivier Danvy. Strong normalization by type-directed partial evaluation and run-time code generation. In Xavier Leroy and Atsushi 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.
6. Vincent Balat, Roberto Di Cosmo, and Marcelo Fiore. Remarks on isomorphisms in typed lambda calculi with empty and sum types. In Gordon D. Plotkin, editor, *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, Copenhagen, Denmark, July 2002. IEEE Computer Society Press. To appear.
7. Gilles Barthe and Olivier Pons. Type isomorphisms and proof reuse in dependent type theory. In Furio Honsell and Marino 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. Ulrich Berger. Program extraction from normalization proofs. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993. Springer-Verlag.
9. Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In Bernhard Möller and John V. Tucker, editors, *Prospects for hardware foundations (NADA)*, number 1546 in Lecture Notes in Computer Science, pages 117–137. Springer-Verlag, 1998.
10. Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In Gilles 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.
11. Kim Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2(2):231–247, 1992.
12. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.
13. Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.
14. Olivier Danvy. Type-directed partial evaluation. In Guy 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.

15. Olivier Danvy. Online type-directed partial evaluation. In Masahiko Sato and Yoshihito Toyama, editors, *Proceedings of the Third Fuji International Symposium on Functional and Logic Programming*, pages 271–295, Kyoto, Japan, April 1998. World Scientific. Extended version available as the technical report BRICS RS-97-53.
16. Olivier Danvy. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter 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.
17. Olivier Danvy, Morten Rhiger, and Kristoffer Rose. Normalization by evaluation with typed abstract syntax. *Journal of Functional Programming*, 11(6):673–680, 2001.
18. Olivier Danvy and René Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. In Kuchen and Swierstra [34], pages 182–197. Extended version available as the technical report BRICS-RS-96-13.
19. David Delahaye, Roberto Di Cosmo, and Benjamin 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*, November 1997.
20. Roberto Di Cosmo. Type isomorphisms in a type assignment framework. In Andrew 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.
21. Roberto Di Cosmo. Deciding type isomorphisms in a type assignment framework. *Journal of Functional Programming*, 3(3):485–525, 1993.
22. Roberto Di Cosmo. *Isomorphisms of types: from λ-calculus to information retrieval and language design*. Birkhauser, 1995. ISBN-0-8176-3763-X.
23. John Doner and Alfred Tarski. An extended arithmetic of ordinal numbers. *Fundamenta Mathematica*, 65:95–127, 1969. See also `http://www.pps.jussieu.fr/~dicosmo/Tarski/`.
24. Andrzej Filinski. A semantic account of type-directed partial evaluation. In Gopalan Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 378–395, Paris, France, September 1999. Springer-Verlag. Extended version available as the technical report BRICS RS-99-17.
25. Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001*, number 2044 in Lecture Notes in Computer Science, pages 151–165, Kraków, Poland, May 2001. Springer-Verlag.
26. Yoshihiko Futamura, Kenroku Nogi, and Akihiko Takano. Essence of generalized partial computation. *Theoretical Computer Science*, 90(1):61–79, 1991.
27. Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
28. Bernd Grobauer. *Topics in Semantics-based Program Manipulation*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, July 2001. BRICS DS-01-6.
29. Bernd Grobauer and Zhe Yang. The second Futamura projection for type-directed partial evaluation. *Higher-Order and Symbolic Computation*, 14(2/3):173–219, 2001.

30. R. Gurevič. Equational theory of positive numbers with exponentiation. *Proceedings of the American Mathematical Society*, 94(1):135–141, May 1985.
31. Simon Helsen and Peter Thiemann. Two flavors of offline partial evaluation. In Jieh Hsiang and Atsushi Ohori, editors, *Advances in Computing Science - ASIAN'98*, number 1538 in Lecture Notes in Computer Science, pages 188–205, Manila, The Philippines, December 1998. Springer-Verlag.
32. John Hughes. Lazy memo-functions. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 129–146, Nancy, France, September 1985. Springer-Verlag.
33. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, London, UK, 1993. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/`.
34. Herbert Kuchen and Doaitse Swierstra, editors. *Eighth International Symposium on Programming Language Implementation and Logic Programming*, number 1140 in Lecture Notes in Computer Science, Aachen, Germany, September 1996. Springer-Verlag.
35. Donald Michie. 'Memo' functions and machine learning. *Nature*, 218:19–22, April 1968.
36. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
37. Larry C. Paulson. *ML for the Working Programmer (2nd edition)*. Cambridge University Press, 1996.
38. Morten Rhiger. *Higher-Order Program Generation*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, July 2001. BRICS DS-01-4.
39. Mikael Rittri. Retrieving library identifiers by equational matching of types. In Mark 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.
40. Mikael Rittri. *Searching program libraries by type and proving compiler correctness by bisimulation*. PhD thesis, University of Göteborg, Göteborg, Sweden, 1990.
41. Mikael Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1):71–89, 1991.
42. Colin Runciman and Ian Toyn. Retrieving re-usable software components by polymorphic type. *Journal of Functional Programming*, 1(2):191–211, 1991.
43. Sergei V. Soloviev. The category of finite sets and cartesian closed categories. *Journal of Soviet Mathematics*, 22(3):1387–1400, 1983.
44. Peter Thiemann. Implementing memoization for partial evaluation. In Kuchen and Swierstra [34], pages 198–212.
45. Jeffrey D. Ullman. *Elements of ML Programming (ML 97 edition)*. Prentice-Hall, 1998.
46. Alex 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).
47. Zhe Yang. *Language Support for Program Generation: Reasoning, Implementation, and Applications*. PhD thesis, Computer Science Department, New York University, New York, New York, August 2001.