Typeful Normalization by Evaluation

Olivier Danvy, Chantal Keller, and Matthias Puech
`{danvy,ckeller,mpuech}@cs.au.dk`

Department of Computer Science, Aarhus University, Denmark

Abstract We present the first typeful implementation of Normalization by Evaluation for the simply typed λ-calculus with sums and control operators:

– we guarantee type preservation and η-long, β-normal forms using only Generalized Algebraic Data Types in a general-purpose programming language, here OCaml; and
– we account for sums and control operators with Continuation-Passing Style.

First, we implement the standard NbE algorithm for the implicational fragment in a typeful way that is correct by construction. We then derive its call-by-value continuation-passing counterpart, that maps a λ-term with sums and call/cc into a CPS term in normal form, which we express in a typed dedicated syntax. Beyond showcasing the expressive power of GADTs, this investigation leads us to deriving, thanks to type inference, novel encodings of the syntax and typing of normal forms, most notably in continuation-passing style.

1 Introduction

A normalization function need not be *reduction-based* and rely on reiterated one-step reduction, according to some strategy, until a normal form is obtained, if any. It can be *reduction-free*, and, as pioneered by Berger and Schwichtenberg [12], one can obtain it by composing an evaluation function (towards a non-standard domain of values) together with a left-inverse reification function (towards normal forms). The concept of this 'normalization by evaluation' (the term is due to Schwichtenberg [11]) arose in a variety of contexts: intuitionistic logic [2,20,43], proof theory [12], program extraction [10], category theory [14,21,46], models of computation [38], program transformation [26], partial evaluation [22,30], etc. [25]. It has been vigorously studied since [1,6,9,40,49].

Normalization by evaluation (NbE for short) has taken a new significance since Grégoire's PhD thesis [35,36] as the reduction engine in the Coq proof assistant [13].[1] It is what makes proofs by reflection practical in Coq today, and in Gonthier's words [34], proofs by reflection are what made it possible to prove the four-colour theorem. There is therefore a renewed interest in NbE today [15].

In this article, we propose a formalization of NbE for the simply-typed λ-calculus with sums and control operators in the general-purpose language OCaml in such a way that the type system guarantees two key properties:

[1] The command `Compute` in Coq triggers a call to Coq's reduction engine.

– NbE produces *normal forms*: the resulting term is in β-short-η-long normal form; and
– NbE is *type-preserving*: if the evaluation function is in direct style, the type of the resulting term is the same as the type of the source term; and if the evaluation function is not in direct style, the two types are not the same but they are related, as developed in Sec. 4.

In Sec. 4, we also extend our formalization to a language with sums and control operators. To this end, we use continuation-passing style (CPS for short), still in a typeful manner. We use CPS not only on terms but also on types to establish that the resulting terms are indeed normal forms in continuation-passing style.

Throughout, we use *Generalized Algebraic Data Types* (GADTs for short), a generalization of ML algebraic data types that allows a fine control on the return type of their constructors [18,47]. We use them to represent the types and the well-typed terms of the simply-typed λ-calculus, and to relate them to the types of values and of normal forms.

Faithful formalizations of NbE in direct style already exist in languages with dependent types like Coq [32]. Instead, we chose a general-purpose programming language with type inference. Our programming language of discourse is OCaml, which now provides support for GADTs [33], but we could have adopted any other functional programming language with this feature, e.g., Haskell. Alternatively, we could have used any other language by encoding GADTs [48] or by using a finally tagless representation of terms [17,42]. Our choice of using GADTs seems simpler in the sense that it enables a methodology where the code is left essentially unchanged, and only the types are refined.

Outline The remainder of this article is an incremental, literate programming exposition of our implementation.[2] We first recall and motivate our starting points: the representation of types, terms, and values in OCaml, the standard NbE algorithm for the implicational fragment in direct style, and GADTs (Sec. 2). We annotate the standard NbE program to obtain a typeful implementation in direct style, that we put to use for the partial evaluation of `printf` directives (Sec. 3). We CPS-transform this typeful implementation, obtaining another typeful implementation that yields typed normal forms in continuation-passing style (Sec. 4). This continuation-passing typeful implementation is ready to be extended with sums and control operators.

2 Background

2.1 Deep and shallow embeddings

Since NbE manipulates types, terms and values of the λ-calculus, we need to represent all of them in our programming language of discourse, OCaml. When embedding a language into another, one has two possibilities: a deep embedding or a shallow embedding.

[2] We will however allow ourselves to reorder code snippets for pedagogical purpose.

In a deep embedding, to each construct of the language corresponds a constructor of a data type; we have access to the structure of terms, and we can define functions over them by structural recursion. The types and terms of the λ-calculus in OCaml can be encoded this way: one data type representing simple types

```
type tp = Base                              (* Some base type *)
        | Arr of tp * tp
```

and another one for terms. For concision, we use a weak (or parametric) HOAS representation of binders [19], where variables belong to an abstract type, and are introduced by a function in OCaml:[3]

```
type tm = Var of x
        | Lam of (x → tm)
        | App of tm * tm
and x (* The variable namespace, uninstantiated for now *)
```

In a shallow embedding, we directly use OCaml constructs to represent constructs in the object language: we lose structural recursion, but we enjoy the property that two equivalent terms in OCaml are indistinguishable. The values of the λ-calculus can be encoded this way: functions are represented as a universal function space, and we reuse OCaml variables and applications syntax nodes. Then, all β-equivalent values are observationally equal.

```
type base (* Some base type, uninstantiated for now *)
type vl = VFun of (vl → vl)
        | VBase of base
```

Example 1. The term $\lambda f x.\, f\ x$ is represented as `Lam (fun f → Lam (fun x → App (Var f, Var x)))` in the deep encoding of terms, and as `VFun (fun (VFun f) → VFun (fun x → f x))` in the shallow encoding of values.

2.2 Normalization by Evaluation

NbE normalizes deeply embedded terms by going through a shallow embedding: an evaluation function maps a deep term to its shallow counterpart, which is then reified back into a deep term. Since $\beta\eta$-convertible terms are indistinguishable at the shallow level, reification has to pick the same representative for two $\beta\eta$-equivalent terms (in practice, the η-long β-normal form, which implies that the result is in normal form).

First, the evaluation function maps deep application nodes into OCaml applications:

```
let rec eval : tm → vl = function
  | Var x → x
```

[3] First-order presentations like de Bruijn indices are also common, and have been showed to be isomorphic to HOAS [5].

```
  | Lam f → VFun (fun x → eval (f x))
  | App (m, n) → match eval m with
    | VFun f → f (eval n)
    | VBase _ → failwith "Unidentified␣Functional␣Object"
```

Variables are substituted with their value in the second case; to this end, we must instantiate their namespace to the type of values, allowing the constructor Var to quote values into terms:

```
and x = vl
```

The expressible values `vl` are shallow, weak head normal forms. The second step consists in reifying them back into an algebraic language of deep terms, or *normal forms* `nf`, that can be inspected by pattern matching:[4]

```
and nf = NLam of (x → nf)
       | NAt of at
and at = AApp of at * nf
       | AVar of x
```

We then define the reification function `reify`, taking a value and its type to a normal form, together with its inverse function, `reflect`. They can be seen as a two-level η-expansion at the given type [28]. This η-expansion stops at base type, which means that values of base type are actually atoms:

```
and base = Atom of at
```

In other words, atoms are the intersection of the set of shallow and deep values, reflecting the fact that values contain both functions and atoms.

All of this leads us to the usual definition of reification and reflection:

```
let rec reify : tp → vl → nf = fun a v → match a, v with
  | Arr (a, b), VFun f → NLam (fun x → reify b (f (reflect a (AVar x))))
  | Base, VBase v → let (Atom r) = v in NAt r
  | _ → failwith "type␣mismatch"
and reflect : tp → at → vl = fun a r → match a with
  | Arr (a, b) → VFun (fun x → reflect b (AApp (r, reify a x)))
  | Base → VBase (Atom r)
```

Finally, NbE maps a term together with its type to a normal form, by composing evaluation and reification:

```
let nbe : tp → tm → nf = fun a m → reify a (eval m)
```

Notice that exceptions might be triggered at runtime if the given term and type do not match. In Section 3, we solve this problem by statically enforcing this match, thanks to GADTs.

[4] To proscribe the representation of β-redexes, we follow the tradition and stratify the syntax into normal forms `nf` (λ-abstractions) and *atoms* `at` (applications).

2.3 GADTs in OCaml

The recent introduction of Generalized Algebraic Data Types [18,47] in OCaml [33] makes it possible to declare data types that are indexed by types, e.g., to write tagless interpreters. Let us illustrate GADTs with the problem of formatting strings *à la* printf in a type-safe way, following Kiselyov [41].

What is the type of the `printf` function in the C programming language? A priori it is dependent: the number of arguments depends on the structure of the first argument, the *formatting directive*. The first author proposed a solution based on polymorphism [23], encoding the formatting directive algebraically as a sequence of literal strings and typed placeholders (written "%d", "%s", etc. in C); GADTs provide a tighter fit. Let us introduce the type of formatting directives, respectively indexed by α, the final type returned by `printf`, and β, the expected type of `printf` when applied only to the directive:

```
type (α, β) directive =
```

These two types coincide when the directive consists only of a literal: no extra argument is then required. We thus explicitly mention the annotation after the argument in the constructor type:

```
  | Lit : string → (α, α) directive
```

When the directive is a placeholder, we add an argument to the expected type of `printf` (these constructors take no arguments):

```
  | String : (α, string → α) directive
  | Int : (α, int → α) directive
```

Finally, the sequence of two directives threads the initial and final types, much like function composition:

```
  | Seq : (β, γ) directive ∗ (α, β) directive → (α, γ) directive
```

After spreading some syntactic sugar, let us try out this definition with an example directive ("%d+%s=%d" in C):

```
let (^^) a b = Seq (a, b) and (!) x = Lit x and d = Int and s = String
let ex_directive : (α, int → string → int → string → α) directive =
  d ^^ !"␣∗␣" ^^ s ^^ !"␣=␣" ^^ d ^^ !"␣in␣" ^^ s
```

Its type reflects the structure of the formatting directive: an integer is expected, and then a string, and then an integer, and then a string, and then the result is whatever it needs to be.

Now, all `printf` needs to do is to map a directive into a usual OCaml primitive function. We first define it in continuation-passing style, and then we apply it to the initial continuation `print_string`, which will emit the formatted string eventually:

```
let rec kprintf : type a b. (a, b) directive → (string → a) → b =
  function
  | Lit s → fun k → k s
  | Int → fun k x → k (string_of_int x)
  | String → fun k x → k (string_of_string x)
  | Seq (f,g) → fun k → kprintf f (fun v → kprintf g (fun w → k (v^w)))
let printf dir = kprintf dir print_string
```

Function `string_of_string` here is the identity. Compared to the previous solution [23], which used one polymorphic function per abstract-syntax constructor of the formatting directive, the dispatch among the constructors is grouped, thanks to GADTs.

Our test directive yields a type-safe printing command:

```
(* prints "6 * 9 = 42 in base 13" *)
let () = printf ex_directive 6 "9" 42 "base␣13"
```

3 Typeful Normalization by Evaluation in Direct Style

Thanks to GADTs, we can decorate the algebraic data types of terms and normal forms with their types, such that only well-typed ones can be represented. This way, the NbE algorithm of Sec. 2.2 can ensure statically that: *i)* no exception is triggered at runtime; *ii)* well-typed terms are mapped to well-typed normal forms; and *iii)* η-long normal forms are produced. We then illustrate this normalizer with a partial evaluator that is guaranteed to preserve the type of the programs it specializes.

3.1 Evaluation

It is a standard use of GADTs to index terms—deep or shallow—by the OCaml type of their interpretation. First, values can be indexed as follows (we will come back to the definition of type `base` later on):

```
type α vl = VFun : (α vl → β vl) → (α → β) vl
          | VBase : base → base vl
```

Similarly for terms:

```
and α x = α vl
type α tm = Lam : (α x → β tm) → (α → β) tm
          | App : (α → β) tm * α tm → β tm
          | Var : α x → α tm
```

The evaluation function now has type α `tm` → α `vl`, ensuring type preservation:

```
let rec eval : type a. a tm → a vl = function
  | Var x → x
  | Lam f → VFun (fun x → eval (f x))
  | App (m, n) → let VFun f = eval m in f (eval n)
```

Because the match between types and terms is ensured statically, there is no need for any exception as in Sec. 2.2. Otherwise, the code is the same.

Remark 1. Evaluation could also have been *tagless*, and thus more efficient [16]: we could have defined directly **type** α `vl` $= \alpha$, but we did not do so to be coherent with Sec. 4. Also, the *finally tagless* approach [17] can alternatively implement typeful NbE without GADTs [42], but that requires significant changes compared to the previous, untyped version: there, evaluation and reification are not recursive functions but define the syntax of terms and types.

3.2 Reification

In the same way, we can index atoms and normal forms with the shallow type of their interpretations:

```
and α nf = NLam : (α x → β nf) → (α → β) nf
         | NAt : base at → base nf
and α at = AApp : (α → β) at ∗ α nf → β at
         | AVar : α x → α at
```

In addition to being β-normal, the restriction of the `at` coercion to a base type guarantees that terms of this data type are also η-long [4].

We then need to statically relate our deep types `tp` with these annotations. To this end, we can index them by the OCaml type of their denotation:

```
type α tp = Base : base tp
          | Arr : α tp ∗ β tp → (α → β) tp
```

The reification function now has type α `tp` $\rightarrow \alpha$ `vl` $\rightarrow \alpha$ `nf`: given a deep type `tp` whose corresponding shallow type is α, and a value of type α `vl`, `reify` yields a normal form α `nf`:

```
let rec reify : type a. a tp → a vl → a nf = fun a v → match a, v with
  | Arr (a, b), VFun f → NLam (fun x → reify b (f (reflect a (AVar x))))
  | Base, VBase v → let (Atom r) = v in NAt r
and reflect : type a. a tp → a at → a vl = fun a r → match a with
  | Arr (a, b) → VFun (fun x → reflect b (AApp (r, reify a x)))
  | Base → VBase (Atom r)
```

As in Sec. 3.1, because the match between types and terms is ensured statically, there is no need for any exception as in Sec. 2.2. Otherwise, the code is the same.

Let us now address the definition of `base`. As before, its values should contain atoms: at base type, terms are interpreted by atoms [32]. But one question remains: what is the type of atoms in the interpretation of the base type? Let us call this type X and let us rely on the implementation as a guideline. In the base case of `reflect`, the type of `r` is refined to `base at`, and the expected type is `base`. Since Atom makes an `base` from an X `at`, we must have $X =$ `base`. Similarly in the base case of `reflect`, the type of `v` is `base`, so `r` has type X `at`, `at r` has type X `nf`. Since the awaited type is `base nf`, we must have $X =$ `base`. The definition of type `base` is thus:

```
and base = Atom of base at
```

This type has no closed inhabitants: they are only constructed and deconstructed during reification and reflection.

Finally, composing evaluation and reification, we obtain typeful NbE, which is guaranteed to map well-typed terms to well-typed normal forms of the same type:

```
let nbe : type a. a tp → a tm → a nf = fun a m → reify a (eval m)
```

It can be read as a cut elimination theorem for intuitionistic logic, apart from termination which is not ensured by OCaml.

3.3 Application: printf, revisited

This section presents an application combining ideas from above: the offline specialization of printf with respect to a formatting directive, using NbE as a partial-evaluation engine. It will specialize the program:

```
fun x y z t →
  printf ex_directive x y z t
```

into the normal form:

```
fun x y z t →
  string_of_int x ^ "␣*␣" ^ y ^ "␣=␣" ^ string_of_int z ^ "␣in␣" ^ t
```

in which ex_directive has been inlined and its processing has been simplified. This specialization is guaranteed to preserve types.

In Sec. 2.3, kprintf was mapping directives to the standard domain of OCaml primitive types. The idea here is to replace the primitive functions (concatenation (^), string_of_int, string_of_string) by a non-standard, syntactic model. By reifying the evaluated program, we obtain a residual term in normal form.

First, we enlarge our representation of atoms (the type α at) with these primitive functions and uninterpreted objects of the types involved (to allow values of different types, we index the type base with a type variable, without consequence on its definition):

```
and α at = (* ... *)
  | APrim : α → α base at
  | AConcat : string base at * string base at → string base at
  | AStringOfInt : int base at → string base at
```

Since we strictly extended the definition of atoms and reify and reflect do not match on them, we can reuse these two functions from Sec. 3.2 as-is.

The primitive functions can now be interpreted as their residual expressions, atoms, instead of as their standard meanings:

```
type int_ = int base at
type string_ = string base at
let string_of_string i = APrim i
let string_of_int x = AStringOfInt x
let (^) s t = AConcat (s, t)
```

The non-standard `printf` is the result of pasting at this point the code from Sec. 2.3, replacing respectively types `int` and `string` by `int_` and `string_`.

Example 2. Let us take this non-standard `printf` function, apply it to our example formatting directive and reify the result at the type of the function:

```
let residual =
  let box f = VFun (fun (VBase (Atom r)) → f r) in
  reify (Arr (Base, Arr (Base, Arr (Base, (Arr (Base, Base))))))
    (box (fun x → box (fun y → box (fun z → box (fun t →
          reflect Base (printf ex_directive x y z t))))))
```

We obtain the specialized program building the final string: `residual` is the normal form mentioned above.

Remark 2. NbE is type-directed, which leads to a completely offline partial evaluator: there is no need to explicitly check at each step of the program whether its result is statically known or not. It differs in that sense from the online partial evaluator proposed by Carette *et al.* [17]. Note that we could nonetheless perform online simplifications in our non-standard primitive functions [24].

4 Typeful Normalization by Evaluation in CPS

In Sec. 3.1, we defined an evaluation function for our object language. It was concise, but leaves no choice of evaluation order or definable control structures: they are inherited from the programming language of discourse, OCaml. In particular, it does not scale seamlessly for disjoint sums and not at all for call/cc:

sums: There is no simple notion of unique normal form for the λ-calculus with sums because of commuting conversions. NbE with sums was nevertheless developed with delimited control operators [22,31,40] and constrained representations of unique normal forms were developed as well [3,7]. Here, we bypass delimited control operators by writing the evaluation function in CPS, and we accept that normal forms are defined modulo commuting conversions.

call/cc: Now that the evaluation function is written in CPS, it is simple to handle call/cc, and the resulting normalization function can immediately be used for programs extracted from classical proofs [27,45].

In this section, we show how to define *typeful* CPS evaluation and reification for the simply-typed λ-calculus with boolean conditionals and call/cc. Our continuation-passing evaluation function maps source terms to continuation-passing values that await a continuation, and allows us to choose the evaluation order and to extend our source language. As in Sec. 3.2, we can then reify these continuation-passing values to a dedicated syntax of normal forms in CPS.

We present the formalization in call by value; the call-by-name variant can be obtained *mutatis mutandis.*

4.1 Typing CPS values

When evaluating in CPS a term of type A, it is well-known [39,44] that its denotation is typed by the CPS-transformed type $\lceil A \rceil$, defined by:

$$\begin{aligned} \lceil A \rceil &= (\lfloor A \rfloor \to o) \to o & \lfloor p \rfloor &= p \\ \lfloor A \to B \rfloor &= \lfloor A \rfloor \to \lceil B \rceil & \lfloor \mathsf{bool} \rfloor &= \mathsf{bool} \end{aligned}$$

where p is a base type, o is the type of answers, and bool is the type of Booleans. The call-by-value transformation can be encoded in the following GADT:

```
type α vl = VFun : (α vl → β md) → (α → β) vl
          | VBase : base → base vl
          | VBool : bool → bool vl
and α md = (α vl → o) → o
```

The type `o` of answers is left unspecified for the moment. Note that the codomain of a function of type $(\alpha \to \beta)$ `vl` expects a continuation (i.e., has type β `md`). For instance, the CPS-transformed identity is written as follows:

```
let id : type a. (a → a) vl = VFun (fun x k → k x)
```

4.2 Evaluation

Let us now extend the syntax of terms with an if statement and with call/cc:

```
type α tm = (* ... *)
  | If  : bool tm * α tm * α tm → α tm
  | CC : ((α → β) → α) tm → α tm
```

Their typing is standard; in particular, call/cc has the type of Peirce's law [37]. Values of type bool are encoded as, e.g., Var (VBool **true**) (remember that we have α `x` $=$ α `vl`).

Now, function `eval` directly maps an α `tm` to an α `md`. Its code can be obtained by CPS-transforming `eval` in Sec. 3.1 with the extra cases:

```
let rec eval : type a. a tm → a md = function
  | Var x → fun c → c x
  | Lam f → fun c → c (VFun (fun x k → eval (f x) k))
  | App (m, n) → fun c → eval m (fun (VFun f) → eval n (fun n → f n c))
  | If (b, m, n) → fun c → eval b (fun (VBool b) →
      if b then eval m c else eval n c)
  | CC m → fun c → eval m (fun (VFun f) → f (VFun (fun x _ → c x)) c)
```

The if case is of no surprise, and could as well have been defined in direct style. The call/cc case captures the continuation `c` into a closure, as customary.

4.3 Reification

Now that the domain of `reify`, i.e., the values α `vl`, is in the image of the CPS transformation, we can CPS-transform the reification function of Sec. 3.2 as well. The types of `reify` and `reflect` will thus be respectively α `tp` $\to \alpha$ `vl` $\to$ (α `nf` $\to$ `o`) $\to$ `o` and α `tp` $\to \alpha$ `at` $\to$ (α `vl` $\to$ `o`) $\to$ `o`. Consequently, the constructor NLam now takes a CPS-transformed function of type α `x` $\to \beta$ `k` $\to$ `o`, where α `k` $= \alpha$ `v` $\to$ `o` and α `v` $= \alpha$ `nf`.

Because of the latter function space, this data type is not a proper weak HOAS. But we can leave types α `k` and α `v` abstract—call these respectively continuation and value variables:

```
type α k and α v
```

and treat the answer type `o` algebraically, i.e., instantiate it by all the operations involving continuation and value variables. There are two of them: applying an α `k` to a normal form in `reify`—call it SRet, and binding a value to an application in `reflect`—call it SBind (previous applications become just value nodes AVal). We are left with the type declarations:

```
and o = SRet : α k * α nf → o
      | SBind : (α → β) at * α nf * (β v → o) → o
and α nf = NLam : (α x → β k → o) → (α → β) nf
         | NAt : base at → base nf
and α at = AVar of α x
         | AVal of α v
```

This typed syntax is in weak HOAS since the domains of functions are abstract. It has in fact been used since the late 1990's [8] to characterize normal forms in CPS. Terms of type `o` are traditionally called *serious terms* after John Reynolds. Note that they do not carry a type like α `nf` and α `at` since they form the type of answers; instead, its constructors act as existentials, linking together types of normal forms, variables and atoms.

Before displaying the code, let us extend the development to Booleans. First, we add the extra case to the type α `tp`:

```
type α tp = (* ... *) | Bool : bool tp
```

Then, we add the constructors and conditional constructs respectively to normal forms and serious terms:

```
and o = (* ... *) | SIf : bool at * o * o → o
and α nf = (* ... *) | NBool : bool → bool nf
```

At last, the full definition of `reify` and `reflect` with Booleans reads:

```
let rec reify : type a. a tp → a vl → (a nf → o) → o =
  fun a v → match a, v with
    | Arr (a, b), VFun f → fun c → c (NLam (fun x k →
        reflect a (AVar x) (fun x → f x (fun v →
            reify b v (fun v → SRet (k, v))))))
    | Base, VBase (Atom r) → fun c → c (NAt r)
    | Bool, VBool b → fun c → c (NBool b)
and reflect : type a. a tp → a at → (a vl → o) → o =
  fun a x → match a, x with
    | Arr (a, b), f → fun c → c (VFun (fun x k →
        reify a x (fun x → SBind (f, x, fun v →
            reflect b (AVal v) (fun v → k v)))))
    | Base, r → fun c → c (VBase (Atom r))
    | Bool, b → fun c → SIf (b, c (VBool true), c (VBool false))
```

Similarly to the direct-style version, it can be seen as a two-level η-expansion, this time performing the expansion rules of CPS with sums [29]. This fact dictates the treatment of conditionals in the last line: they are serious terms, and duplicate the context c in their two branches.

As an epilogue, we compose evaluation and reification to obtain normalization. Unlike the direct case, where a value was reified into a normal form, a CPS value is reified as a serious term abstracted by an initial continuation. At top level, NbE in CPS thus returns such an abstraction:

```
type α c = Init of (α k → o)
let nbe : type a. a tp → a tm → a c = fun a m →
  Init (fun k → eval m (fun m → reify a m (fun v → SRet (k, v))))
```

5 Summary and Future Work

We have presented the first typeful implementation of NbE for the simply-typed λ-calculus in the minimalistic setting of a general-purpose programming language with GADTs. To the best of our knowledge, our implementation is the first one to ensure by typing that its output is not only in β-normal form, but also in η-long form. We have illustrated how NbE achieves partial evaluation by specializing a typeful version of `printf` with respect to any given formatting directive. By CPS-transforming our typeful implementation, we have obtained systematically the syntax and typing rules of normal forms in continuation-passing style. Finally, we have presented the first typeful implementation of NbE for the simply-typed λ-calculus with sums and control operators in the same minimalistic setting. This normalization function can be used for programs extracted from classical proofs, and the resulting normal form can then be mapped back to direct style.

Future work includes developing a version of NbE that is parameterized by an arbitrary monad (i.e., not just the identity monad or a continuation monad). In this version, the non-standard evaluation function will be monadic. Monadic reification with effect preservation seems like a tall order, but given a monad, reification towards a (well-typed but non-monadic) normal form seems in sight: it could be achieved using the type transformation associated to this given monad; a monadic version of the direct-style transformation would then be necessary to map this non-monadic normal form to a monadic normal form. Such a monadic version of NbE would make it possible to normalize programs whose effects can be described with monads, e.g., probabilistic or stateful computations.

References

1. A. Abel. Towards normalization by evaluation for the $\beta\eta$-calculus of constructions. In M. Blume and G. Vidal, editors, *FLOPS*, number 6009 in LNCS, pages 224–239, Sendai, Japan, Apr. 2010. Springer. 1
2. A. Abel, T. Coquand, and P. Dybjer. Normalization by evaluation for Martin-Löf type theory with typed equality judgements. In J. Marcinkowski, editor, *LICS*, pages 3–12, Wroclaw, Poland, July 2007. IEEE Computer Society Press. 1
3. T. Altenkirch, P. Dybjer, M. Hofmann, and P. J. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In J. Halpern, editor, *LICS*, pages 203–210, Boston, Massachusetts, June 2001. IEEE Computer Society Press. 9
4. T. Altenkirch, M. Hofmann, and T. Streicher. Categorical reconstruction of a reduction-free normalization proof. In D. H. Pitt, D. E. Rydeheard, and P. Johnstone, editors, *CTCS*, number 953 in LNCS, pages 182–199, Cambridge, UK, Aug. 1995. Springer. 7
5. R. Atkey, S. Lindley, and J. Yallop. Unembedding domain-specific languages. In S. Weirich, editor, *Haskell*, pages 37–48. ACM, 2009. 3
6. V. Balat. *Une étude des sommes fortes: isomorphismes et formes normales.* PhD thesis, PPS, Université Denis Diderot (Paris VII), Paris, France, Dec. 2002. 1
7. V. Balat, R. D. Cosmo, and M. P. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In X. Leroy, editor, *POPL*, SIGPLAN Notices, Vol. 39, No. 1, pages 64–76, Venice, Italy, Jan. 2004. ACM Press. 9
8. V. Balat and O. Danvy. Strong normalization by type-directed partial evaluation and run-time code generation. In X. Leroy and A. Ohori, editors, *TIC*, number 1473 in LNCS, pages 240–252, Kyoto, Japan, Mar. 1998. Springer. 11
9. F. Barral. *Decidability for non standard conversions in typed λ-calculus.* PhD thesis, Ludvig-Maximilians-Universität and Université Paul Sabatier, München, Germany and Toulouse, France, June 2008. 1
10. U. Berger. Program extraction from normalization proofs. In M. Bezem and J. F. Groote, editors, *TLCA*, number 664 in LNCS, pages 91–106, Utrecht, The Netherlands, Mar. 1993. Springer. 1
11. U. Berger, M. Eberl, and H. Schwichtenberg. Normalization by evaluation. In B. Möller and J. V. Tucker, editors, *NADA*, number 1546 in LNCS, pages 117–137, Berlin, Germany, 1998. Springer. 1
12. U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In G. Kahn, editor, *LICS*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press. 1
13. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development.* Springer, 2004. 1
14. I. Beylin and P. Dybjer. Extracting a proof of coherence for monoidal categories from a proof of normalization for monoids. In S. Berardi and M. Coppo, editors, *TYPES*, number 1158 in LNCS, pages 47–61, Torino, Italy, June 1995. Springer. 1
15. M. Boespflug. *Conception d'un noyau de vérification de preuves pour le $\lambda\Pi$-calcul modulo.* PhD thesis, École Polytechnique, Palaiseau, France, Jan. 2011. 1

16. M. Boespflug, M. Dénès, and B. Grégoire. Full reduction at full throttle. In J.-P. Jouannaud and Z. Shao, editors, *CPP*, number 7086 in LNCS, pages 362–377, Kenting, Taiwan, Dec. 2011. Springer. 7
17. J. Carette, O. Kiselyov, and C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009. 2, 7, 9
18. J. Cheney and R. Hinze. First-class phantom types. Tech. Report 1901, Computing and Information Science, Cornell University, Ithaca, New York, 2003. 2, 5
19. A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In J. Hook and P. Thiemann, editors, *ICFP*, SIGPLAN Notices, Vol. 43, No. 9, pages 143–156, Victoria, British Columbia, Sept. 2008. ACM Press. 3
20. T. Coquand and P. Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997. 1
21. D. Čubrić, P. Dybjer, and P. J. Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, 8:153–192, 1998. 1
22. O. Danvy. Type-directed partial evaluation. In G. L. Steele Jr., editor, *POPL*, pages 242–257, St. Petersburg Beach, Florida, Jan. 1996. ACM Press. 1, 9
23. O. Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6):621–625, 1998. 5, 6
24. O. Danvy. Online type-directed partial evaluation. In M. Sato and Y. Toyama, editors, *FLOPS*, pages 271–295, Kyoto, Japan, Apr. 1998. World Scientific. 9
25. O. Danvy and P. Dybjer, editors. *Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation (NBE 1998)*, BRICS Note Series NS-98-8, Gothenburg, Sweden, May 1998. Available online at `http://www.brics.dk/~nbe98/programme.html`. 1, 15
26. O. Danvy and A. Filinski. Abstracting control. In M. Wand, editor, *LFP*, pages 151–160, Nice, France, June 1990. ACM Press. 1
27. O. Danvy and J. L. Lawall. Back to direct style II: First-class continuations. In W. Clinger, editor, *LFP*, LISP Pointers, Vol. V, No. 1, pages 299–310, San Francisco, California, June 1992. ACM Press. 9
28. O. Danvy, K. Malmkjær, and J. Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(3):209–227, 1995. 4
29. O. Danvy, K. Malmkjær, and J. Palsberg. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems*, 8(6):730–751, 1996. 12
30. P. Dybjer and A. Filinski. Normalization and partial evaluation. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics – Advanced Lectures*, number 2395 in LNCS, pages 137–192, Caminha, Portugal, Sep. 2000. Springer. 1
31. A. Filinski. Normalization by evaluation for the computational lambda-calculus. In S. Abramsky, editor, *TLCA*, number 2044 in LNCS, pages 151–165, Kraków, Poland, May 2001. Springer. 9
32. F. Garillot and B. Werner. Simple types in type theory: Deep and shallow encodings. In K. Schneider and J. Brandt, editors, *TPHOLs*, number 4732 in LNCS, pages 368–382, Kaiserslautern, Germany, Sept. 2007. Springer. 2, 7
33. J. Garrigue and D. Rémy. Ambivalent types for principal type inference with GADTs. In C. c. Shan, editor, *APLAS*, volume 8301 of *LNCS*, pages 257–272. Springer, 2013. 2, 5
34. G. Gonthier. Formal proof – the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, Dec. 2008. 1
35. B. Grégoire. *Compilation de Termes de Preuves, un (nouveau) mariage entre Coq et OCaml.* PhD thesis, Université Denis Diderot (Paris 7), Paris, France, 2003. 1

36. B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In S. Peyton Jones, editor, *ICFP*, SIGPLAN Notices, Vol. 37, No. 9, pages 235–246, Pittsburgh, Pennsylvania, Sept. 2002. ACM Press. 1
37. T. G. Griffin. A formulae-as-types notion of control. In P. Hudak, editor, *POPL*, pages 47–58, San Francisco, California, Jan. 1990. ACM Press. 10
38. P. Hancock. The model of computable terms. In Danvy and Dybjer [25]. Available online at `http://www.brics.dk/~nbe98/programme.html`. 1
39. J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In H.-J. Boehm, editor, *POPL*, pages 458–471, Portland, Oregon, Jan. 1994. ACM Press. 10
40. D. Iliḱ. *Constructive Completeness Proofs and Delimited Control*. PhD thesis, École Polytechnique, Palaiseau, France, Oct. 2010. Winner of Kurt Gödel Research Prize Fellowship 2010. 1, 9
41. O. Kiselyov. Type-safe functional formatted IO, 2008. Web post, available at `http://okmij.org/ftp/typed-formatting/`. 5
42. O. Kiselyov. Typed Tagless Final Interpreters. In J. Gibbons, editor, *SSGIP*, volume 7470 of *LNCS*, pages 130–174. Springer, 2010. 2, 7
43. P. Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In *Proceedings of the Third Scandinavian Logic Symposium (1972)*, volume 82 of *Studies in Logic and the Foundation of Mathematics*, pages 81–109. North-Holland, 1975. 1
44. A. R. Meyer and M. Wand. Continuation semantics in typed lambda-calculi (summary). In R. Parikh, editor, *Logics of Programs*, number 193 in LNCS, pages 219–224, Brooklyn, New York, June 1985. Springer. 10
45. C. R. Murthy. *Extracting Constructive Content from Classical Proofs*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, 1990. 9
46. J. C. Reynolds. Normalization and functor categories. In Danvy and Dybjer [25]. Available online at `http://www.brics.dk/~nbe98/programme.html`. 1
47. H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In G. Morrisett, editor, *POPL*, SIGPLAN Notices, Vol. 38, No. 1, pages 224–235, New Orleans, Louisiana, Jan. 2003. ACM Press. 2, 5
48. J. Yallop and O. Kiselyov. First-class modules: hidden power and tantalizing promises. Presented at the 2010 Workshop on ML, 2010. 2
49. Z. Yang. *Language Support for Program Generation: Reasoning, Implementation, and Applications*. PhD thesis, Computer Science Department, New York University, New York, New York, Aug. 2001. 1