Abstract

We present the first typeful implementation of Normalization by Evaluation for the simply typed \(\lambda\)-calculus with sums and control operators: we guarantee type preservation and \(\eta\)-long (modulo commuting conversions), \(\beta\)-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.

Our presentation takes the form of a typed functional pearl. First, we implement the standard NbE algorithm for the implicational fragment in a typeful way that is correct by construction. We then derive its continuation-passing counterpart, in call-by-value and call-by-name, that maps a \(\lambda\)-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, we emphasize that type inference gives a smooth way to re-derive the encodings of the syntax and typing of normal forms in Continuation-Passing Style.

1998 ACM Subject Classification D.1.1 Applicative (Functional) Programming, F.3.2 Semantics of Programming Languages

Keywords and phrases Normalization by Evaluation, Generalized Algebraic Data Types, Continuation-Passing Style, partial evaluation

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

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 [15], 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 [14]) arose in a variety of contexts: intuitionistic logic [1, 21, 48], proof theory [15], program extraction [12], category theory [22, 54], models of computation [40], program transformation [28], partial evaluation [24, 33], etc. [27]. It has been vigorously studied since [8, 11, 43, 57].

A recent example of the power of normalization by evaluation (NbE for short) lies in the new reduction engine developed by Boespflug et al. [16, 17] for the Coq proof assistant.\(^1\) It

\(^1\) The command Compute in Coq triggers a call to Coq’s reduction engine.
improves the efficiency of proofs by reflection by an order of magnitude [4], and in Gonthier’s words [38], proofs by reflection are what made it possible to prove the four-color theorem.

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

- NbE produces normal forms: the resulting term is in $\eta$-long, $\beta$-normal form;
- NbE is type-preserving: the type of the resulting term is the same as the type of the source term.

These are guaranteed by OCaml’s subject reduction, provided that we stay in its purely functional, terminating fragment (which is a meta-argument).

To address sums and control operators, we use Continuation-Passing Style (CPS for short) in a novel way: we show that CPS-transforming the standard, typed NbE algorithm not only leaves room for these constructs, but also lets us derive a syntax of CPS normal forms and its typing rules. The resulting NbE program maps typed $\lambda$-terms to typed CPS normal forms.

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 [19, 55]. We use them not only to represent the types and the well-typed terms of the simply-typed $\lambda$-calculus, but more interestingly to relate them to the types of values and of normal forms. The use of GADTs inherently limits us to simply typed objects languages, but our main motivation is to give a clean presentation of NbE for non-trivial aspects of such languages.

Faithful formalizations of NbE in direct style already exist in languages with dependent types like Coq or Agda [6, 13, 36, 42]. These complex languages already rely on an implementation of normalization for type-checking, which is precisely what we embark on implementing. Instead, we chose a general-purpose programming language featuring only weak-head evaluation and type inference. Our programming language of discourse is OCaml, which now provides support for GADTs [37], but we could have adopted any other functional programming language with this feature, e.g., Haskell, as partly done by Danvy, Rhiger, and Rose with type classes [32]. (We write “partly” because the “long” aspect of the resulting $\eta$-long, $\beta$-normal forms needed a meta-argument.) Alternatively, we could have used any other functional language by encoding GADTs [56] or by using some indirect representation of terms as functions (“finally tagless”, phantom types, etc.) [18, 47]. Using GADTs, we can keep representing syntax as algebraic data types, as customary. This conservative design 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 in the form of a typed functional pearl. 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 (Section 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 (Section 3). We CPS-transform this typeful implementation, obtaining another typeful implementation that yields typed normal forms in CPS (Section 4). This continuation-passing typeful implementation is ready to be extended with sums and control operators.

---

2 We will however allow ourselves to pedagogically reorder some code snippets. The full code is currently available at cs.mcgill.ca/~puech/typeful.ml.
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 essentially two options: a deep embedding or a shallow embedding.

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 can be encoded this way in OCaml: one data type representing simple types (featuring an uninstantiated base type)

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

and another one for terms. For concision, we use a weak (or parametric) Higher-Order Abstract Syntax representation of binders [20] (HOAS for short), where variables belong to an abstract type, and are introduced by OCaml functions:

```ocaml
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 βη-equivalent values in OCaml are observationally equal. 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.

```ocaml
type base (* Base type, uninstantiated for now *)
type vl =
| VFun of (vl -> vl)
| VBase of base
```

Example 1. The term \( \lambda x. f \ x \) is represented as \( \text{Lam} \ (\text{fun} \ f \rightarrow \text{Lam} \ (\text{fun} \ x \rightarrow \text{App} \ (\text{Var} \ f, \text{Var} \ x))) \) in the deep encoding of terms, and as \( \text{VFun} \ (\text{fun} \ (\text{VFun} \ f) \rightarrow \text{VFun} \ (\text{fun} \ x \rightarrow f \ x)) \) 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 βη-equivalent terms are indistinguishable at the shallow level, reification has to

---

3 First-order presentations like de Bruijn indices are also common, and have been showed to be isomorphic to weak HOAS [7]. This way, we avoid Kripke-like parametrization of the target language, and we separate concerns better.
pick the same representative for two \( \beta \eta \)-equivalent terms (in practice, the \( \eta \)-long \( \beta \)-normal form, which implies that the result is in normal form).

First, the evaluation function maps application nodes \texttt{App} in the deep encoding into shallow, OCaml applications:

```ocaml
let rec eval : tm \rightarrow vl = function
| Var x \rightarrow x
| Lam f \rightarrow VFun (fun x \rightarrow eval (f x))
| App (m, n) \rightarrow match eval m with
  | VFun f \rightarrow f (eval n)
  | VBase b \rightarrow failwith "Unidentified_Functional_Object"
```

In the second case, variables are substituted with their value; to this end, we must instantiate their namespace with the type of values, allowing the constructor \texttt{Var} to quote values into terms:

```ocaml
and x = vl
```

The expressible values \( vl \) are shallow values, i.e., 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:

```ocaml
and nf =
| NLam of (y \rightarrow nf)
| NAt of at
and at =
| AApp of at * nf
| AVar of y
and y
```

To proscribe the representation of \( \beta \)-redexes, we follow the tradition and stratify the syntax into normal forms \( nf \) (\( \lambda \)-abstractions) and atoms \( at \) (applications). Type \( y \) is the uninstantiated domain of target variables.

We then define the reification function \texttt{reify}, taking a value and its type to a normal form, together with its inverse function, \texttt{reflect}. They can be seen as performing a two-level \( \eta \)-expansion at the given type [30]. This \( \eta \)-expansion stops when encountering a value of the uninstantiated base type, which means that values of base type actually stand for atoms:

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

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

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

```ocaml
let rec reify : tp \rightarrow vl \rightarrow nf = fun a v \rightarrow match a, v with
| Arr (a, b), VFun f \rightarrow NLam (fun x \rightarrow reify b (f (reflect a (AVar x))))
| Base, VBase v \rightarrow let (Atom r) = v in NAt r
| a, v \rightarrow failwith "type_mismatch"

and reflect : tp \rightarrow at \rightarrow vl = fun a r \rightarrow match a with
| Arr (a, b) \rightarrow VFun (fun x \rightarrow reflect b (AApp (r, reify a x)))
```

---

4 One could object that this instantiation of the domain of variables takes us away from weak HOAS. However, it is only necessary for the source language of \texttt{eval}, and a commodity to avoid more verbose solutions like de Bruijn indices or explicit parametricity in type \( x \) [51].
Finally, NbE maps a term together with its type to a normal form, by composing evaluation and reification:

```ocaml
let nbe : tp \to tm \to nf = fun a m \to 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.

### 2.3 GADTs in OCaml

The recent introduction of Generalized Algebraic Data Types [19, 55] in OCaml [37] makes it syntactically possible to constrain type parameters for the return type of the constructors of a data type, which enables, 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 [46] and OCaml’s recent Printf module; it will serve as a running example in this article.

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 [25], encoding the formatting directive algebraically as a sequence of literal strings and typed placeholders (written "%d", "%s", etc. in C) and encoding it with CPS. GADTs provide language support for this encoding. Let us introduce the type of formatting directives, respectively indexed by $\alpha$, the final type returned by `printf`, and $\beta$, the expected type of `printf` when applied only to the directive:

```ocaml
type (\alpha, \beta) 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:

```ocaml
| Lit : string \to (\alpha, \alpha) directive
```

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

```ocaml
| String : (\alpha, string \to \alpha) directive
| Int : (\alpha, int \to \alpha) directive
```

Finally, the sequence of two directives threads the initial and final types, much like function composition (and indeed the first author’s encoding for sequence was function composition in CPS):

```ocaml
| Seq : (\beta, \gamma) directive * (\alpha, \beta) directive \to (\alpha, \gamma) directive
```

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

```ocaml
let (^) a b = Seq (a, b) and (!) x = Lit x and d = Int and s = String
let ex_directive : (\alpha, int \to string \to int \to string \to \alpha) directive =
    d ^ !"%d" ^ !"%s" ^ !"%i" ^ !"%d" ^ !"%in" ^ !"%s"
```
The 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 CPS, and then we apply it to the initial continuation `print_string`, which will emit the formatted string eventually:

```ocaml
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 solutions [5, 25], which used one polymorphic function per abstract-syntax constructor of the formatting directive, the dispatch among the constructors is grouped, thanks to the GADTs.

Our test directive yields a type-safe printing command:

```ocaml
(* prints "6 * 9 = 42 in base 13" *)
let () = printf ex_directive 6 "9" 42 "base/uni242313"
```

### 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 Section 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 (in addition to being β-normal, which is new [32]). 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):

```ocaml
type α vl =
| VFun : (α vl -> β vl) -> (α -> β) vl
| VBase : base -> base vl
```

Note that this type definition does not respect the positivity condition, in the sense of, e.g., Coq, because there is a negative occurrence of `vl`. It is, however, stratified in the sense of Abella [35], i.e., its type parameter gets syntactically smaller. Thus, it forms a valid inductive definition. Ditto for terms (the same remark as in Section 2.2 applies to type `α x`):

```ocaml
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 \( \alpha \text{tm} \rightarrow \alpha \text{vl} \), ensuring type preservation:

```ocaml
let rec eval : type a. a tm \rightarrow a vl = function
  | Var x \rightarrow x
  | Lam f \rightarrow VFun (fun x \rightarrow eval (f x))
  | App (m, n) \rightarrow 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 Section 2.2. Otherwise, the code remains the same.

**Remark.** Evaluation could also have been *tagless*, and thus more efficient [17], i.e., we could have defined directly type \( \alpha \text{vl} = \alpha \). We did not do so to be consistent with Section 4. Also, the *finally tagless* approach [18] can alternatively implement typeful NbE without GADTs [47], but it 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 type of their interpretations:

```ocaml
and \( \alpha \text{nf} = \)
  | NLam : (\( \alpha \text{y} \rightarrow \beta \text{nf} \)) \rightarrow (\( \alpha \rightarrow \beta \)) \text{nf}
  | NAt : base \text{at} \rightarrow base \text{nf}
and \( \alpha \text{at} = \)
  | AApp : (\( \alpha \rightarrow \beta \)) \text{at} \ast \( \alpha \text{nf} \rightarrow \beta \text{at} \)
  | AVar : \( \alpha \text{y} \rightarrow \alpha \text{at} \)
and \( \alpha \text{y} \)
```

The variable domain \( \alpha \text{y} \) is left uninstantiated. In addition to being \( \beta \)-normal, the restriction of the \text{NAt} coercion to a base type guarantees that terms of this data type are also \( \eta \)-long [3].

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

```ocaml
type \( \alpha \text{tp} = \)
  | Base : base \text{tp}
  | Arr : \( \alpha \text{tp} \ast \beta \text{tp} \rightarrow (\( \alpha \rightarrow \beta \)) \text{tp} \)
```

The reification function now has type \( \alpha \text{tp} \rightarrow \alpha \text{vl} \rightarrow \alpha \text{nf} \): given a deep type \( \text{tp} \) whose corresponding shallow type is \( \alpha \), and a value of type \( \alpha \text{vl} \), \text{reify} yields a normal form of type \( \alpha \text{nf} \):

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

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

Let us now address the definition of \text{base}. As before, its values should contain atoms: at base type, terms are interpreted by atoms [36]. 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 \( \text{base at} \), and the expected type is \( \text{base} \). Since \( \text{Atom} \) makes a \( \text{base} \) from an \( X \) at, we must have \( X = \text{base} \). Similarly in the base case of reify, the type of \( v \) is \( \text{base} \), so \( r \) has type \( X \) at, \( \text{Nat} r \) has type \( X \) \( \text{nf} \). Since the awaited type is \( \text{base} \) \( \text{nf} \), we must have \( X = \text{base} \). The definition of type \( \text{base} \) is thus:

\[
\begin{align*}
\text{and base} &= \text{Atom of base at}
\end{align*}
\]

This type has no (normalizing) closed inhabitants: they are only constructed and deconstructed during reification and reflection. Its definition is faithful to previous formalizations, where the interpretation of the base type is the set of atomic terms at base type.

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

\[
\begin{align*}
\text{let nbe : type a. a tp } &\rightarrow \text{ a tm } \rightarrow \text{ a nf } = \text{ fun a m } \rightarrow \text{ reify a (eval m)}
\end{align*}
\]

This function can be read as a cut elimination for intuitionistic logic, apart from termination which is not ensured by OCaml, but is a meta-argument: all three functions \( \text{eval} \), \( \text{reify} \) and \( \text{reflect} \) are defined by structural induction over their first argument.

### 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. Given the same formatting directive as in Section 2.3, the program

\[
\begin{align*}
\text{fun x y z t } &\rightarrow \text{ printf ex_directive x y z t}
\end{align*}
\]

is specialized into the normal form

\[
\begin{align*}
\text{fun x y z t } &\rightarrow \text{ string_of_int x } ^\text{/uni2423} \text{ */uni2423} ^\text{/uni2423} \text{/uni2423} ^\text{/=uni2423} ^\text{string_of_int z } ^\text{/uni2423} \text{in/uni2423} ^\text{/uni2423} \text{t}
\end{align*}
\]

in which \( \text{ex_directive} \) has been inlined and part of its processing has been carried out. This specialization is guaranteed to preserve types.

In Section 2.3, \( \text{kprintf} \) was mapping directives to the standard domain of OCaml primitive types. The idea here is to replace the primitive functions (concatenation \( ^\text{/uni2423} \), \( \text{string_of_int} \), \( \text{string_of_string} \)) 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 \( \alpha \) at) with these primitive functions and uninterpreted objects of the types involved (to allow values of different types, we index the type \( \text{base} \) with a type variable, without consequence on its definition):

\[
\begin{align*}
\text{and } \alpha \text{ at } &= \langle * \ldots * \rangle \\
| \text{APrim } : \alpha &\rightarrow \alpha \text{ base at} \\
| \text{AConcat } : \text{string base at} &\times \text{string base at } \rightarrow \text{string base at} \\
| \text{AStringOfInt } : \text{int base at } &\rightarrow \text{string base at}
\end{align*}
\]

Since we strictly extended the definition of atoms and \( \text{reify} \) and \( \text{reflect} \) do not match on them, we can reuse these two functions from Section 3.2 as they are.

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

\[
\begin{align*}
\text{type int_ } &= \text{ int base at} \\
\text{type string_ } &= \text{ string base at} \\
\text{let string_of_string } i &= \text{APrim } i
\end{align*}
\]
let string_of_int x = AStringOfInt x
let (^) s t = AConcat (s, t)

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

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 (this can ben witnessed by pretty-printing it, or converting it to a de Bruijn representation [7]).

Remark. 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. [18]. Note that we could nonetheless perform online simplifications in our non-standard primitive functions [26].

4 Typeful Normalization by Evaluation in CPS

In Section 3.1, we defined an evaluation function for our object language. It is 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 \(\lambda\)-calculus with sums because of commuting conversions [43]. NbE with sums was nevertheless developed with delimited control operators [24, 34, 43] and constrained representations of unique normal forms were developed as well [2, 9]. Here, we bypass delimited control operators by writing the evaluation function in CPS, and we accept that normal forms are defined modulo commuting conversions (our notion of \(\eta\)-expansion is thus limited by them).

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 [29, 50].

In this section, we show how to define typeful CPS evaluation and reification for the simply-typed \(\lambda\)-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 Section 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 first (Sections 4.1 to 4.3), and then just sketch the call-by-name variant (Section 4.4).

\[^5\] Another choice could have been \texttt{shift} and \texttt{reset}, as Ilik did in Coq [44].
4.1 Typing CPS values

When evaluating in CPS a term of type $A$, it is well-known \[49\] that its denotation is typed by the CPS-transformed type $\lceil A \rceil$, defined by:

\[
\begin{align*}
\lceil A \rceil &= (\lfloor A \rfloor \rightarrow o) \rightarrow o \\
\lceil A \rightarrow B \rceil &= \lceil A \rceil \rightarrow \lceil B \rceil \\
\lceil \text{bool} \rceil &= \text{bool}
\end{align*}
\]

where $p$ is an (uninstantiated) base type, $o$ is the type of answers, and \text{bool} is the type of Booleans. The call-by-value transformation can be reflected in the GADT that encodes CPS-values:

\[
\text{type } \alpha \ 	ext{vl =}
| \ VFun : (\alpha \ 	ext{vl} \rightarrow \beta \ 	ext{md}) \rightarrow (\alpha \rightarrow \beta) \ 	ext{vl} \\
| \ VBase : \text{base} \rightarrow \text{base} \ 	ext{vl} \\
| \ VBool : \text{bool} \rightarrow \text{bool} \ 	ext{vl} \\
\text{and } \alpha \ 	ext{md} = (\alpha \ 	ext{vl} \rightarrow o) \rightarrow o
\]

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

\[
\begin{align*}
\text{let app : type } a \ b. ( (a \rightarrow b) \rightarrow a \rightarrow b ) \ 	ext{vl =}
& \ VFun \ ( \text{fun} \ ( \ VFun \ f) \ k \rightarrow k \ ( \ VFun \ ( \text{fun} \ x \ k \rightarrow f \ x \ ( \text{fun} \ v \rightarrow k \ v))) )
\end{align*}
\]

4.2 Evaluation

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

\[
\text{type } \alpha \ 	ext{tm = (* ... *)}
| \ \text{If} : \text{bool} \ 	ext{tm} \star \alpha \ 	ext{tm} \star \alpha \ 	ext{tm} \rightarrow \alpha \ 	ext{tm} \\
| \ \text{CC} : ((\alpha \rightarrow \beta) \rightarrow \alpha) \ 	ext{tm} \rightarrow \alpha \ 	ext{tm}
\]

Their typing is standard; call/cc has the type of Peirce’s law \[39\]. Values of type \text{bool} are encoded as, e.g., \text{Var} (\text{VBool \ true}) (remember that $\alpha \ x = \alpha \ 	ext{vl}$).

Now, function eval directly maps an $\alpha \ 	ext{tm}$ to an $\alpha \ 	ext{md}$. Its code can be obtained by CPS-transforming eval in Section 3.1 with the extra cases:

\[
\begin{align*}
\text{let rec eval : type } a. a \ 	ext{tm} \rightarrow a \ 	ext{md} = \text{function} \\
& \ Var \ x \rightarrow \text{fun} \ c \rightarrow c \ x \\
& \ \text{Lam} \ f \rightarrow \text{fun} \ c \rightarrow c \ ( \ VFun \ ( \text{fun} \ x \ k \rightarrow \text{eval} \ (f \ x) \ k) ) \\
& \ \text{App} \ (m, n) \rightarrow \text{fun} \ c \rightarrow \text{eval} \ m \ ( \ VFun \ f ) \rightarrow \text{eval} \ n \ ( \text{fun} \ n \rightarrow f \ n \ c) \\
& \ \text{If} \ (b, m, n) \rightarrow \text{fun} \ c \rightarrow \text{eval} \ b \ ( \text{fun} \ \text{VBool} \ b) \rightarrow \\
& \ \text{if} \ b \ \text{then} \ \text{eval} \ m \ c \ \text{else} \ \text{eval} \ n \ c \\
& \ \text{CC} \ m \rightarrow \text{fun} \ c \rightarrow \text{eval} \ m \ ( \ VFun \ f ) \rightarrow \text{f} \ ( \ VFun \ ( \text{fun} \ x \ k \rightarrow c \ x)) \ c)
\end{align*}
\]

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 function, as customary in Scheme.

4.3 Reification

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

Because of the latter function space, this data type is not a proper weak HOAS. But we can leave types \( \alpha \text{k} \) and \( \alpha \text{v} \) abstract—call these respectively continuation and value variables (\( \alpha \text{y} \) is the domain of source variables):

```plaintext
type \( \alpha \text{k} \) and \( \alpha \text{v} \) and \( \alpha \text{y} \)
```

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

```plaintext
and \( \text{o} = \)
| \text{SRet} : \( \alpha \text{k} \times \alpha \text{nf} \rightarrow \text{o} \)
| \text{SBind} : (\( \alpha \rightarrow \beta \)) \text{at} \times \alpha \text{nf} \times (\beta \text{v} \rightarrow \text{o}) \rightarrow \text{o} \)
and \( \alpha \text{nf} = \)
| \text{NLam} : (\( \alpha \text{y} \rightarrow \beta \text{k} \rightarrow \text{o} \)) \rightarrow (\( \alpha \rightarrow \beta \)) \text{nf}
| \text{NAt} : \text{base} \rightarrow \text{base} \text{nf}
and \( \alpha \text{at} = \)
| \text{AVar of} \( \alpha \text{y} \)
| \text{AVal of} \( \alpha \text{v} \)
```

This typed syntax is in weak HOAS since the domains of variables are abstract. It has in fact been used since the late 1990’s [10] to characterize normal forms in CPS: terms of type \( \text{o} \) are traditionally called serious terms after Reynolds [52], and represent computations. Note that they do not carry a type like \( \alpha \text{nf} \) and \( \alpha \text{at} \) since they form the type of answers; instead, its constructors act as existentials, linking together types of normal forms, variables and atoms, and hiding them away. Normal forms are traditionally called trivial terms, again after Reynolds [52].

Before displaying the code, let us extend the development to Booleans. First, we add the extra case to the type \( \alpha \text{tp} \):

```plaintext
and \( \text{o} = \)
| \text{SIf} : \text{bool} \times \alpha \text{at} \times \text{o} \rightarrow \text{o} \)
and \( \alpha \text{nf} = \)
| \text{NBool} : \text{bool} \rightarrow \text{bool} \text{nf}
```

Then, we add Booleans and conditional expressions to normal forms and serious terms, respectively:

```plaintext
and \( \text{o} = (\ast \ ... \ast) \)
| \text{SIf} : \text{bool} \times \alpha \text{at} \times \text{o} \rightarrow \text{o} \)
and \( \alpha \text{nf} = (\ast \ ... \ast) \)
| \text{NBool} : \text{bool} \rightarrow \text{bool} \text{nf}
```

At last, the full definition of \text{reify} and \text{reflect} with Booleans reads:

```plaintext
let rec \text{reify} : type \( a \). \( a \text{tp} \rightarrow a \text{vl} \rightarrow (a \text{nf} \rightarrow \text{o}) \rightarrow \text{o} = \)
| \text{Arr} (a, b), \text{VFun} f \rightarrow \text{fun} c \rightarrow c (\text{NLam} (\text{fun} x k \rightarrow \text{reflect} a (\text{AVar} x) (\text{fun} x \rightarrow f x (\text{fun} v \rightarrow \text{reify} b v (\text{fun} v \rightarrow \text{SRet} (k, v)))))\))
| \text{Base}, \text{VBase} (\text{Atom} r) \rightarrow \text{fun} c \rightarrow c (\text{NAt} r)
| \text{Bool}, \text{VBool} b \rightarrow \text{fun} c \rightarrow c (\text{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, these two functions can be seen as performing a two-level
η-expansion, this time with the expansion rules of CPS with sums [31]. This fact dictates the
treatment of conditionals in the last line: they are serious terms, and duplicate the context c
in their two branches.

We can now compose evaluation and reification to obtain normalization. A CPS value is
reified as a program in normal form: a serious term abstracted by its initial continuation.
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))))

As an epilogue, we strip out the resulting syntax of its type annotations to obtain the
familiar syntax of call-by-value CPS normal forms:

\[ P ::= \lambda k. S \]
\[ S ::= k T | R S (\lambda v. S) | if(R, S, S) \]
\[ T ::= \lambda yk. S | true | false | R \]
\[ R ::= y | v \]

As in the direct-style case, it is syntactically impossible to form a redex in this syntax, thanks
to the stratification of trivial terms and atoms.

4.4 In call by name

In call by name, the domains of functions are also computations (i.e., expecting a continuation),
as presented in Section 4.1. This transformation is reflected:

- in the type of values in that functions now expect a continuation:
  
  ```
  type α vl = (* ... *)
  | VFun : (α md → β md) → (α → β) vl
  ```

- in the variables of the source language that now range over thunks instead of values:
  
  ```
  and α x = α md
  ```

- and in the variables of the target language: they are now serious terms, and are associated
  with a continuation binding their values; for the same reason, the argument to a “bind”
  is now a thunk:

  ```
  and o =
  | SRet : α k * α nf → o
  | SBind : (α → β) at * (α k → o) * (β v → o) → o
  | SIf : bool at * o * o → o
  | SVar : α y * (α v → o) → o
  ```

  ```
  and α nf =
  ```
Evaluation and reification functions are modified *mutatis mutandis*:

```ocaml
let rec eval : 'a tm -> 'a md = function
| Var x c -> x c
| Lam f c -> c (VFun (fun x k -> eval (f x) k))
| App (m, n) c -> c (eval m (fun (VFun f) k -> f (eval n) k))
| Bool b c -> c (VBool b)
| If (b, m, n) c -> eval b (function VBool true -> eval m c | VBool false -> eval n c)
| CC m c -> eval m (fun (VFun f) k -> f (eval k) c)

let rec reify : 'a tp -> 'a vl -> (a nf -> o) -> o = fun a v -> match a, v with
| Arr (a, b), f c -> c (VFun (fun x k ->
  (fun v k -> SVar (y, fun v -> reflect a (AVal v) k))
  (fun v k -> reify b v (fun v k -> SRet (k, v)))))
| Bool, b c -> c (VBool b)
| Base, r c -> c (VBase (Atom r))

and reflect : 'a tp -> 'a at -> (a vl -> o) -> o = fun a x -> match a, x with
| Arr (a, b), f c -> c (VFun (fun x k ->
  SBind (f, (fun x k -> (fun v k -> reify a v (fun v k -> SRet (k, v))))
    (fun v k -> reify b v (fun v k -> SRet (k, v)))))
| Bool, b c -> c (VBool true, c (VBool false))
| Base, r c -> c (VBase (Atom r))
```

As before, these two functions can be seen as performing a two-level \(\eta\)-expansion, this time with the expansion rules of call-by-name CPS [41].

We can finally compose evaluation and reification to obtain normalization. As in the call-by-value case, NbE in call-by-name CPS returns a program, i.e., a serious term abstracted by the initial continuation:

```ocaml
let nbe : 'a tp -> 'a tm -> (a nf -> o) -> o = fun a m k -> eval m (fun m -> reify a m k)
```

As an epilogue, we strip out the resulting syntax of its type annotations to obtain the familiar syntax of call-by-name CPS normal forms:

```ocaml
P ::= \lambda k. \mathsf{S} \quad \text{Programs}
S ::= \mathsf{k} \mathsf{T} | R (\lambda k. \mathsf{S}) (\lambda v. \mathsf{S}) | \mathsf{if}(R, S, S) | y (\lambda v. S)
T ::= \lambda y k. \mathsf{S} \quad \text{Serious terms}
R ::= \mathsf{true} | \mathsf{false} | R \quad \text{Trivial terms}
\mathsf{v} \quad \text{Atoms}
```

Again, it is syntactically impossible to form a redex in this syntax.
Summary and Future Work

We have presented the first typeful implementation of NbE for the simply-typed $\lambda$-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 $\beta$-normal form, but also in $\eta$-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 CPS. Finally, we have presented the first typeful implementation of NbE for the simply-typed $\lambda$-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 [23, 29].

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 is 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


4 Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In Jouannaud and Shao [45], pages 135–150.


7 Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding domain-specific languages. In Stephanie Weirich, editor, *Proceedings of the 2nd ACM SIGPLAN Symposium on


Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. In Jouannaud and Shao [45], pages 362–377.


