Typeful Normalization by Evaluation

Olivier Danvy[1], Chantal Keller[2], and Matthias Puech[3]

1 **Department of Computer Science, Aarhus University, Denmark**
`danvy@cs.au.dk`

2 **Microsoft Research Cambridge, United Kingdom and**
Microsoft Research – Inria Joint Centre, France
`Chantal.Keller@inria.fr`

3 **School of Computer Science, McGill University, Montreal, Canada**
`puech@cs.mcgill.ca`

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 (modulo commuting conversions), β-normal forms using only Generalized Algebraic Data Types in a general-purpose programming language, here OCaml; and
- we account for finite 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 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, 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.TYPES.2014.72

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, 53], models of computation [40], program transformation [28], partial evaluation [24, 33], etc. [27]. It has been vigorously studied since [8, 11, 43, 56].

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.

20th International Conference on Types for Proofs and Programs (TYPES 2014).
Editors: Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau; pp. 72–88
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 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:

- NbE produces *normal forms*: the resulting term is in η-long, β-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 λ-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, 54]. We use them not only to represent the types and the well-typed terms of the simply-typed λ-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 η-long, β-normal forms needed a meta-argument.) Alternatively, we could have used any other functional language by encoding GADTs [55] 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.[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 (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)

```
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:[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 $\beta\eta$-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.

```
type base                            (* 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 $\beta\eta$-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 η-long β-normal form, which implies that the result is in normal form).

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

```
let rec eval : tm → vl = function
  | Var x → x
  | Lam f → VFun (fun x → eval (f x))
  | App (m, n) → match eval m with
    | VFun f → f (eval n)
    | VBase b → 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 Var to quote values into terms:[4]

```
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:

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

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). Type `y` is the uninstantiated domain of target variables.

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

```
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:

```
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
  | a, v → failwith "type␣mismatch"
```

[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 `eval`, and a commodity to avoid more verbose solutions like de Bruijn indices or explicit parametricity in type `x` [51].

```
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.

2.3 GADTs in OCaml

The recent introduction of Generalized Algebraic Data Types [19, 54] 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 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 (and indeed the first author's encoding for sequence was function composition in CPS):

```
  | 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␣%s"` 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
```

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:

```
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:

```
(* 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 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):

```
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`):

```
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` $\rightarrow$ 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 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` α `vl =` α. 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:

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

The variable domain α `y` is left uninstantiated. In addition to being β-normal, the restriction of the `NAt` coercion to a base type guarantees that terms of this data type are also η-long [3].

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` $\rightarrow$ α `vl` $\rightarrow$ α `nf`: given a deep type `tp` whose corresponding shallow type is α, and a value of type α `vl`, `reify` yields a normal form of type 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 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 `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 `base at`, and the expected type is `base`. Since Atom makes a `base` from an X `at`, we must have $X =$ `base`. Similarly in the base case of `reify`, the type of `v` is `base`, so `r` has type X `at`, NAt `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 (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:

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

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 `eval`, `reify` and `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

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

is specialized 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 part of its processing has been carried out. This specialization is guaranteed to preserve types.

In Section 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 Section 3.2 as they are.

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 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 λ-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 η-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].[5]

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 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 shift and 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:

$$\lceil A \rceil = (\lfloor A \rfloor \to o) \to o \qquad \lfloor p \rfloor = p$$
$$\lfloor A \to B \rfloor = \lfloor A \rfloor \to \lceil B \rceil \qquad \lfloor \mathsf{bool} \rfloor = \mathsf{bool}$$

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

```
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 `(α → β) vl` expects a continuation (i.e., has type `β md`). For instance, the CPS-transformed applicator is written as follows:

```
let app : type a b. ((a → b) → a → b) vl =
  VFun (fun (VFun f) k → k (VFun (fun x k → f x (fun v → k v))))
```

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; call/cc has the type of Peirce's law [39]. Values of type bool are encoded as, e.g., `Var (VBool true)` (remember that `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 Section 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 k → 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 function, as customary in Scheme.

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 Section 3.2 as well. The types of `reify` and `reflect` will thus be respectively α `tp` $\to$ α `vl` $\to$ `(`α `nf` $\to$ `o)` $\to$ `o` and α `tp` $\to$ α `at` $\to$ `(`α `vl` $\to$ `o)` $\to$ `o`. Consequently, the constructor `NLam` now takes a CPS-transformed function of type α `y` $\to$ β `k` $\to$ `o`, where α `k` = α `v` $\to$ `o` and α `v` = α `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 (α `y` is the domain of source variables):

```
type α k and α v and α y
```

We treat the answer type `o` algebraically, i.e., we 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 just become 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 y → β k → o) → (α → β) nf
  | NAt : base at → base nf
and α at =
  | AVar of α y
  | AVal of α 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 `o` are traditionally called *serious terms* after Reynolds [52], and represent computations. 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, 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 α `tp`:

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

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

```
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, 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:

$P ::= \lambda k.\, S$	Programs
$S ::= k\, T \mid R\, S\, (\lambda v.\, S) \mid \mathsf{if}(R, S, S)$	Serious terms
$T ::= \lambda yk.\, S \mid \mathsf{true} \mid \mathsf{false} \mid R$	Trivial terms
$R ::= y \mid v$	Atoms

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 =
```

```
  | NLam : (α y → β k → o) → (α → β) nf
  | NBool : bool → bool nf
  | NAt : base at → base nf
and α at =
  | AVal : α v → α at
```

Evaluation and reification functions are modified *mutatis mutandis*:

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

```
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 y k →
        f (fun k → SVar (y, fun v → reflect a (AVal v) k))
          (fun v → reify b v (fun v → SRet (k, v)))))
  | Bool, VBool b → fun c → c (NBool b)
  | Base, VBase (Atom r) → fun c → c (NAt r)

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 →
      SBind (f, (fun k → x (fun v → reify a v (fun v → SRet (k, v)))),
             (fun v → reflect b (AVal v) k))))
  | Bool, b → fun c → SIf (b, c (VBool true), c (VBool false))
  | Base, r → fun c → c (VBase (Atom r))
```

As before, these two functions can be seen as performing a two-level η-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:

```
let nbe : type a. 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:

$P ::= \lambda k.\, S$	Programs
$S ::= k\, T \mid R\,(\lambda k.\, S)\,(\lambda v.\, S) \mid \mathsf{if}(R, S, S) \mid y\,(\lambda v. S)$	Serious terms
$T ::= \lambda yk.\, S \mid \mathsf{true} \mid \mathsf{false} \mid R$	Trivial terms
$R ::= v$	Atoms

Again, it is syntactically impossible to form a redex in this syntax.

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 CPS. 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 [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.

Acknowledgments. This work was carried out while the two last authors were visiting the first, in the fall of 2013. We are grateful to our anonymous reviewers for their helpful comments.

References

1. Andreas Abel, Thierry Coquand, and Peter Dybjer. Normalization by evaluation for Martin-Löf type theory with typed equality judgements. In Jerzy Marcinkowski, editor, *Proceedings of the 22nd IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 3–12, Wroclaw, Poland, July 2007. IEEE Computer Society Press.
2. Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip J. 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.
3. 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.
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.
5. Kenichi Asai. On typing delimited continuations: three new solutions to the printf problem. *Higher-Order and Symbolic Computation*, 22(3):275–291, 2009.
6. Kenichi Asai, Luminous Fennell, Peter Thiemann, and Yang Zhang. A type theoretic specification for partial evaluation. In Olaf Chitil, Andy King, and Olivier Danvy, editors, *Proceedings of the 16th ACM SIGPLAN International Conference on Principles and*

Practice of Declarative Programming (PPDP'14), pages 57–68, Canterbury, UK, September 2014. ACM Press.

7 Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding domain-specific languages. In Stephanie Weirich, editor, *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, pages 37–48. ACM, 2009.

8 Vincent Balat. *Une étude des sommes fortes: isomorphismes et formes normales.* PhD thesis, PPS, Université Denis Diderot (Paris VII), Paris, France, December 2002.

9 Vincent Balat, Roberto Di Cosmo, and Marcelo P. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In Xavier Leroy, editor, *Proceedings of the Thirty-First Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 39, No. 1, pages 64–76, Venice, Italy, January 2004. ACM Press.

10 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.

11 Freiric 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.

12 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.

13 Ulrich Berger, Stefan Berghofer, Pierre Letouzey, and Helmut Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82(1):25–49, 2006.

14 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, Berlin, Germany, 1998. Springer-Verlag.

15 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.

16 Mathieu Boespflug. *Conception d'un noyau de vérification de preuves pour le λΠ-calcul modulo.* PhD thesis, École Polytechnique, Palaiseau, France, January 2011.

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

18 Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009.

19 James Cheney and Ralf Hinze. First-class phantom types. Technical Report 1901, Computing and Information Science, Cornell University, Ithaca, New York, 2003.

20 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In James Hook and Peter Thiemann, editors, *Proceedings of the 2008 ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, SIGPLAN Notices, Vol. 43, No. 9, pages 143–156, Victoria, British Columbia, September 2008. ACM Press.

21 Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.

22 Djordje Čubrić, Peter Dybjer, and Philip J. Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, 8:153–192, 1998.

23 Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.

24 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.

25 Olivier Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6):621–625, 1998.

26 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.

27 Olivier Danvy and Peter Dybjer, editors. *Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation (NBE 1998)*, BRICS Note Series NS-98-8, Gothenburg, Sweden, May 1998. BRICS, Department of Computer Science, Aarhus University. Available online at `http://www.brics.dk/~nbe98/programme.html`.

28 Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.

29 Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In William Clinger, editor, *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, pages 299–310, San Francisco, California, June 1992. ACM Press.

30 Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(3):209–227, 1995.

31 Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems*, 8(6):730–751, 1996.

32 Olivier Danvy, Morten Rhiger, and Kristoffer Rose. Normalization by evaluation with typed abstract syntax. *Journal of Functional Programming*, 11(6):673–680, 2001.

33 Peter Dybjer and Andrzej Filinski. Normalization and partial evaluation. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics – Advanced Lectures*, number 2395 in Lecture Notes in Computer Science, pages 137–192, Caminha, Portugal, September 2000. Springer-Verlag.

34 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.

35 Andrew Gacek, Dale Miller, and Gopalan Nadathur. A two-level logic approach to reasoning about computations. *Journal of Automated Reasoning*, 49(2):241–273, 2012.

36 François Garillot and Benjamin Werner. Simple types in type theory: Deep and shallow encodings. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007*, number 4732 in Lecture Notes in Computer Science, pages 368–382, Kaiserslautern, Germany, September 2007. Springer.

37 Jacques Garrigue and Didier Rémy. Ambivalent types for principal type inference with gadts. In Chung-chieh Shan, editor, *Programming Languages and Systems – 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*, volume 8301 of *Lecture Notes in Computer Science*, pages 257–272. Springer, 2013.

38 George Gonthier. Formal proof – the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, December 2008.

39 Timothy G. Griffin. A formulae-as-types notion of control. In Paul Hudak, editor, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, January 1990. ACM Press.

40 Peter Hancock. The model of computable terms. In Danvy and Dybjer [27]. Available online at http://www.brics.dk/~nbe98/programme.html.

41 John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. *Journal of Functional Programming*, 7(3):303–319, 1997.

42 Noriko Hirota and Kenichi Asai. Formalizing a correctness property of a type-directed partial evaluator. In Nils Anders Danielsson and Bart Jacobs, editors, *Proceedings of the 2014 ACM SIGPLAN Workshop on Programming Languages meets Program Verification, PLPV 2014, January 21, 2014, San Diego, California, USA, Co-located with POPL'14*, pages 41–46. ACM, 2014.

43 Danko Ilik. *Constructive Completeness Proofs and Delimited Control*. PhD thesis, École Polytechnique, Palaiseau, France, October 2010.

44 Danko Ilik. A formalized type-directed partial evaluator for shift and reset. In Ugo de'Liguoro and Alexis Saurin, editors, *Proceedings of the First Workshop on Control Operators and their Semantics, COS 2013, Eindhoven, The Netherlands, June 24-25, 2013*, volume 127 of *EPTCS*, pages 86–100, 2013.

45 Jean-Pierre Jouannaud and Zhong Shao, editors. *Certified Programs and Proofs – First International Conference, CPP 2011*, number 7086 in Lecture Notes in Computer Science, Kenting, Taiwan, December 2011. Springer.

46 Oleg Kiselyov. Type-safe functional formatted IO, 2008. Web post, available at http://okmij.org/ftp/typed-formatting/.

47 Oleg Kiselyov. Typed tagless final interpreters. In Jeremy Gibbons, editor, *Generic and Indexed Programming – International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, volume 7470 of *Lecture Notes in Computer Science*, pages 130–174. Springer, 2010.

48 Per 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.

49 Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logics of Programs – Proceedings*, number 193 in Lecture Notes in Computer Science, pages 219–224, Brooklyn, New York, June 1985. Springer.

50 Chetan R. Murthy. *Extracting Constructive Content from Classical Proofs*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, 1990.

51 Matthias Puech. Parametric HOAS with first-class modules. https://syntaxexclamation.wordpress.com/2014/06/27/parametric-hoas-with-first-class-modules/, 2014.

52 John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972.

53 John C. Reynolds. Normalization and functor categories. In Danvy and Dybjer [27]. Available online at http://www.brics.dk/~nbe98/programme.html.

54 Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In Greg Morrisett, editor, *Proceedings of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 38, No. 1, pages 224–235, New Orleans, Louisiana, January 2003. ACM Press.

55 Jeremy Yallop and Oleg Kiselyov. First-class modules: hidden power and tantalizing promises. Presented at the 2010 Workshop on ML, 2010.

56 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.