Normalization by Evaluation Does *not* Need Types

Klaus Aehlig*

Abstract

A variant of normalization by evaluation is presented that works for weakly normalizing terms of the *untyped* λ-calculus. The main tool is defining the (semantical) application in such a way that applying a syntactical object to an arguments performs the application by bringing the argument into the syntax as well.

1 Introduction and related work

Normalization by evaluation [3] is usually presented as follows: Take an appropriate semantics for the typed λ-calculus and define evaluation "as usual". Because $\beta\eta$-equality holds in the "real world", normalization is already performed by evaluation. The task then is to get a syntactical representation out of the semantical object. This is usually achieved by defining two functionals $\uparrow_\rho$ and $\downarrow^\rho$ (also called "make self evaluating"/"procedure to object" or "reflect"/"reify") for every type *by induction on the type*.

Although this approach has been generalized [1, 6] to polymorphic types and system F or by adding constants [2] the restriction to typed terms and the requirement of knowing the type still seems to be a major restriction to the use of normalization by evaluation.

However, the main argument for normalization by evaluation to work can be formulated *without* the use of types: semantics is invariant under β-reduction and for normal terms evaluating and then reifying reproduces. This was the motivation for the version of normalization by evaluation proposed in this paper which works also for the *untyped* λ-calculus.

*Mathematisches Institut, Ludwig-Maximilians-Universität München, Theresienstraße 39, 80333 München, Germany; Tel +49 89 2394 4415, `aehlig@rz.mathematik.uni-muenchen.de`.
Supported by "Graduiertenkolleg Logik in der Informatik" of the "Deutsche Forschungsgemeinschaft".

The new technical idea is the following: if we don't have types, we might be faced with the situation that syntax is applied to some function; in this case we take the argument into the syntax as well (via reify) and then perform a *syntactical* application.

We show (in subsection 2.3) correctness with respect to a termmodel obtained by identifying terms that have a common reduct with respect to a reduction relation describing the behaviour of the functions "application", "reify" and the properties of the (semantical) λ-binder. Corollary 2.24 states that $\downarrow(\llbracket t \rrbracket)$ is the normal form of t in this model, if t has a normal form. This leads (in section 3) to a simple algorithm for normalizing untyped λ-terms *without* the need of defining a substitution.

Besides being more flexible, this approach might also give some insight in the behaviour of the $\downarrow(\cdot)$ functional: "If we have a function, we state this by a syntactical λ and try what the function does when applied to a new variable".

The paper is organized as follows: In section 2 we present the system as a "two-level λ-calculus" in the style of [6], in section 3 we discuss additional problems that have to be solved for an implementation of this system and finally in section 4 we conclude.

2 Presentation as two-level λ-Terms

We first describe our variant of normalization by evaluation as a term rewriting system in the style of a two-level λ-calculus and show that it has the desired properties. This way of presenting normalization by evaluation is due to [6]. An implementation of this system is given in section 3.

2.1 Syntax

Definition 2.1 (terms). The set Λ of two-level λ-Terms is inductively defined:

$$\Lambda \ni r, s \ ::= \ \underline{x} \mid \overline{x} \mid \underline{\lambda \underline{x}}\, r \mid \overline{\lambda \overline{x}}\, r \mid r \underline{@} s \mid r \overline{@} s \mid \downarrow(r)$$

Overlined abstraction and application should be understood as the "real" ones, i.e. we will have β-reduction for them. In the implementation they correspond to the function abstraction of the implementation language. Underlined abstraction and application should be understood as syntactical ones, i.e. these are the syntactical entities we whish to talk about. The $\downarrow(\cdot)$ construct is the analogon of the "reify" function in the present framework. As mentioned in the introduction we need this function to define the correct behavior of the semantical application.

We identify terms that only differ in the naming of (over- *and* underlined) bound variables; in the implementation the renaming of *overlined* variables would be handled by the binding mechanism of the implementation language, whereas for the *underlined* variables we use de Bruijn indices (see section 3.1), so that this problem does not occur.

Definition 2.2. A term $t \in \Lambda$ is called *syntactical* if it contains no overlining.

Definition 2.3 (reductions). Define the relations $\mapsto_\beta$, $\mapsto_{\mathtt{app}}$ and $\mapsto_{\mathtt{down}}$ as follows:

$$
\begin{array}{llll}
(\overline{\lambda x}\, t)\overline{@}s & \mapsto_\beta & t\,[\overline{x}{:=}s] & \\[1em]
t\overline{@}s & \mapsto_{\mathtt{app}} & t\underline{@}{\downarrow}(s) & t \text{ syntactical} \\[1em]
{\downarrow}(\overline{\lambda x}\, s) & \mapsto_{\mathtt{down}} & \underline{\lambda z}\, {\downarrow}(s\,[\overline{x}{:=}\underline{z}]) & \underline{z} \text{ “new''} \\[1em]
{\downarrow}(t) & \mapsto_{\mathtt{down}} & t & t \text{ syntactical}
\end{array}
$$

The closure of $\mapsto_\beta$, $\mapsto_{\mathtt{app}}$, $\mapsto_{\mathtt{down}}$ under all term forming operations is denoted by $\to_\beta$, $\to_{\mathtt{app}}$, $\to_{\mathtt{down}}$ respectively.

Remark 2.4. $\underline{z}$ new means that the variable $\underline{z}$ is not captured by any (syntactical) abstraction in $s\,[\overline{x}{:=}\underline{z}]$. As where identifying α-equal terms, this is a well-defined notation in our theoretical treatment; as mentioned above in the implementation the problem does not occur due to the use of de Bruijn indices.

Remark 2.5. The first rule of $\mapsto_{\mathtt{down}}$ can be motivated by specializing the “typed version” ${\downarrow}^{\rho\to\sigma}\,(t) = \underline{\lambda z}\,{\downarrow}^\sigma\,(t\overline{@}{\uparrow}_\rho\,\underline{z})$ to t being of the form $\overline{\lambda x}\,s$, omitting the ${\uparrow}_\rho$ due to lack of knowledge of the necessary η-expansions and carrying out the newly generated β redex.

2.2 Confluence

In this section we will show that in the calculus above $\to_\beta\ \cup \to_{\mathtt{app}}\ \cup \to_{\mathtt{down}}$ is confluent. In particular, all normal forms are unique, if they exist and in particular all syntactical ${\downarrow}(\cdot)$-free reducts are unique (as they are normal). The modular presentation of the proof is inspired by [5].

Proposition 2.6. *For any substitution θ not mapping syntactical variables to non-syntactical terms, if $t \to_{\mathtt{down}} t'$ then $t\theta \to_{\mathtt{down}} t'\theta$.*

Proof. Almost immediate from definition, noting that the restriction to the substitution θ ensures that $r\theta$ is syntactical if r is and using known facts about commutation of substitutions. ∎

Lemma 2.7 ($\diamond$-property of $\to_{\mathtt{down}}$). *If $t'\ {}_{\mathtt{down}}\!\leftarrow t \to_{\mathtt{down}} t''$ then there is a term t''' such that $t' \to_{\mathtt{down}} t'''\ {}_{\mathtt{down}}\!\leftarrow t''$.*

Proof. We argue by induction on t. The only non-trivial case is $\downarrow\!(\overline{\lambda \overline{x}\, s'}) \;{}_{\mathtt{down}}\!\leftarrow \downarrow\!(\overline{\lambda \overline{x}\, s}) \;\rightarrow_{\mathtt{down}}\; \underline{\lambda z}\; \downarrow\!(s\,[\overline{x}{:=}\underline{z}])$, with $s \;\rightarrow_{\mathtt{down}}\; s'$. Here, due to proposition 2.6 the common one-step reduct is $\underline{\lambda z}\; \downarrow\!(s'\,[\overline{x}{:=}\underline{z}])$. ∎

Lemma 2.8 ($\diamond$-property of $\rightarrow_{\mathsf{app}}$). *If $t' \;{}_{\mathsf{app}}\!\leftarrow\; t \;\rightarrow_{\mathsf{app}}\; t''$ then there is a term t''' such that $t' \;\rightarrow_{\mathsf{app}}\; t''' \;{}_{\mathsf{app}}\!\leftarrow\; t''$.*

Proof. Here, the only non-trivial case is $t\underline{@}\!\downarrow\!(s) \;{}_{\mathsf{app}}\!\leftarrow\; t\overline{@}s \;\rightarrow_{\mathsf{app}}\; t\overline{@}s'$ with common one-step reduct $t\underline{@}\!\downarrow\!(s')$. ∎

Lemma 2.9 ($\rightarrow_{\mathtt{down}}$ and $\rightarrow_{\mathsf{app}}$ commute). *If $t' \;{}_{\mathsf{app}}\!\leftarrow\; t \;\rightarrow_{\mathtt{down}}\; t''$ then there is a term t''' such that $t' \;\rightarrow_{\mathtt{down}}\; t''' \;{}_{\mathsf{app}}\!\leftarrow\; t''$*

Proof. There are two non-trivial cases to consider, depending on whether $\rightarrow_{\mathtt{down}}$ or $\rightarrow_{\mathsf{app}}$ is the outer reduction. In the first case we have $t\underline{@}\!\downarrow\!(s) \;{}_{\mathsf{app}}\!\leftarrow\; t\overline{@}s \;\rightarrow_{\mathtt{down}}\; t\overline{@}s'$ with the common reduct $t\underline{@}\!\downarrow\!(s')$. In the second case we have $\downarrow\!(\overline{\lambda \overline{x}\, s'}) \;{}_{\mathsf{app}}\!\leftarrow \downarrow\!(\overline{\lambda \overline{x}\, s}) \;\rightarrow_{\mathtt{down}}\; \underline{\lambda x}\, s\,[\overline{x}{:=}\underline{z}]$ with common reduct $\underline{\lambda x}\; \downarrow\!(s'\,[\overline{x}{:=}\underline{z}])$, using proposition 2.6 again. ∎

Definition 2.10 (syntactical reduction). Define $\rightarrow_{\mathsf{s}} \;=\; \rightarrow_{\mathsf{app}} \;\cup\; \rightarrow_{\mathtt{down}}$, i.e. $t \;\rightarrow_{\mathsf{s}}\; t'$ if and only if $t \;\rightarrow_{\mathsf{app}}\; t'$ or $t \;\rightarrow_{\mathtt{down}}\; t'$.

Corollary 2.11. *$\rightarrow_{\mathsf{s}}$ has the $\diamond$-property and therefore in particular is confluent.*

This should not come to a surprise to us, as $\rightarrow_{\mathsf{app}}$ and $\rightarrow_{\mathtt{down}}$ more or less just remove underlinings and therefore no real overlap occurs.

Proposition 2.12. *For any substitution θ not mapping syntactical variables to non-syntactical terms, if $t \;\rightarrow_{\mathsf{app}}\; t'$ then $t\theta \;\rightarrow_{\mathsf{app}}\; t'\theta$.*

Proof. Immediate from definition, noting that the restriction to the substitution θ ensures that $r\theta$ is syntactical if r is. ∎

The next lemma shows the main property of our reduction relations: $\rightarrow_{\beta}$ and $\rightarrow_{\mathsf{app}}$ do not infer, as in $t\overline{@}s \;\rightarrow_{\mathsf{app}}\; t\underline{@}s$, t has to be syntactical and therefore cannot be of the form $\overline{\lambda\cdot}\ldots$, i.e. we do not loose a β-redex.

Lemma 2.13. $t' \;{}_{\mathsf{app}}\!\leftarrow\; t \;\rightarrow_{\beta}\; r'' \quad\Rightarrow\quad \exists t'''.\, t' \;\rightarrow_{\beta}\; t''' \;{}^{*}_{\mathsf{app}}\!\leftarrow\; t''$

Proof. We distinguish cases according to the relative positions of the redexes. If they do not overlap the claim is trivial. If the $\rightarrow_{\beta}$-redex is more outside and the $\rightarrow_{\mathsf{app}}$-redex is in the argument, this redex will be copied any number of times (including possibly 0), but is unchanged; this is why the reflexive-transitive closure is needed. If the $\rightarrow_{\beta}$-redex is more outside, but the $\rightarrow_{\mathsf{app}}$-redex is in the function, we use proposition 2.12, noting that the reduction due to a $\rightarrow_{\beta}$-redex only subsitutes semantical variables.

If the $\rightarrow_{\text{app}}$ redex is more outside, where in the situation $t\overline{@}s \rightarrow_{\text{app}} t\underline{@}{\downarrow}(s)$. Here t has to be syntactical, therefore the $\rightarrow_\beta$ -redex has to be within s and we trivially have the common reduct $t\underline{@}{\downarrow}(s')$. □

Lemma 2.14. $t' \ {}_{\text{down}}\!\leftarrow \ t \ \rightarrow_\beta \ r'' \quad \Rightarrow \quad \exists t'''.\, t' \ \rightarrow_\beta \ t''' \ {}^{*}_{\text{down}}\!\leftarrow \ t''$

Proof. We again distinguish according to the relative position of the redexes. If the $\rightarrow_\beta$ -redex is more outside, we argue as above, using proposition 2.6.

If the $\rightarrow_{\text{app}}$ -redex is more outside, we cannot be in the situation ${\downarrow}(t) \rightarrow_{\text{down}} t$, since t would have to be syntactical and therefore cannot host a $\rightarrow_\beta$ -redex. So we have to consider ${\downarrow}(\overline{\lambda \overline{x}}\, s') \ {}_\beta\!\leftarrow {\downarrow}(\overline{\lambda \overline{x}}\, s) \ \rightarrow_{\text{down}} \ \underline{\lambda \underline{z}}\ {\downarrow}(s\,[\overline{x}{:=}\underline{z}])$. Here $\underline{\lambda \underline{x}}\ {\downarrow}(s'\,[\overline{x}{:=}\underline{z}])$ is a common reduct, using that $\rightarrow_\beta$ commutes with substitution. □

The next two lemmata show that in a reduction sequence, the $\rightarrow_\beta$ reductions can be done first.

Lemma 2.15. $\rightarrow_{\text{app}}\ \rightarrow_\beta \ \subset\ \rightarrow_\beta\ \rightarrow^{*}_{\text{app}}$ *, i.e. if* $t \ \rightarrow_{\text{app}} \ t' \ \rightarrow_\beta \ t''$*, then* $t \ \rightarrow_\beta \ t''' \ \rightarrow^{*}_{\text{app}} \ t''$ *for some term* t'''*.*

Proof. Trivial case distinction acording to position of the $\rightarrow_\beta$ -redex relative to the position of the $\rightarrow_{\text{app}}$ -redex. □

Lemma 2.16. $\rightarrow_{\text{down}}\ \rightarrow_\beta \ \subset\ \rightarrow_\beta\ \rightarrow^{*}_{\text{down}}$

Proof. If the $\rightarrow_{\text{down}}$ -reduction is of the form ${\downarrow}(t) \ \rightarrow_{\text{app}} \ t$, the $\rightarrow_\beta$ -reduction has to be outside (since t is syntactical) and possible dupplication of the $\rightarrow_{\text{down}}$ -redex might occur, or we have to use proposistion 2.6.

Otherwise the $\rightarrow_{\text{down}}$ redex has to be of the form ${\downarrow}(\overline{\lambda \overline{x}}\, s) \ \rightarrow_{\text{down}} \ \underline{\lambda \underline{z}}\ {\downarrow}(s\,[\overline{x}{:=}\underline{z}])$. The only new case is, if the $\rightarrow_\beta$ -reduction occurs inside, i.e. if $s\,[\overline{x}{:=}\underline{z}] \ \rightarrow_\beta \ s'$. Then s' has to be of the form $s' = s''$ with $s \ \rightarrow_\beta \ s''$, hence ${\downarrow}(\overline{\lambda \overline{x}}\, s) \ \rightarrow_\beta {\downarrow}(\overline{\lambda \overline{x}}\, s'') \ \rightarrow_{\text{app}} \ \underline{\lambda \underline{z}}\ {\downarrow}(s''\,[\overline{x}{:=}\underline{z}])$. □

Corollary 2.17. $(\ \rightarrow_\beta \ \cup \ \rightarrow_{\text{s}}\ )^* = \ \rightarrow^{*}_\beta\ \rightarrow^{*}_{\text{s}}$

Lemma 2.18. $t' \ {}^{*}_{\text{s}}\!\leftarrow \ t \ \rightarrow_\beta \ t'' \quad \Rightarrow \quad \exists t'''.\, t' \ \rightarrow_\beta \ t''' \ {}^{*}_{\text{s}}\!\leftarrow \ t''.$

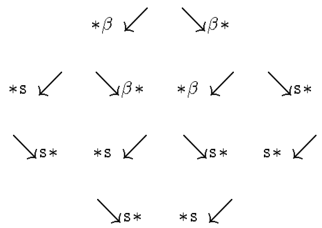
Proof. Induction on the number of steps in $t' \ {}^{*}_{\text{s}}\!\leftarrow \ t$, using lemma 2.13 or lemma 2.14 and afterwards the induction hypothesis. □

Lemma 2.19. $t'' \ {}^{*}_{\text{s}}\!\leftarrow \ t \ \rightarrow^{*}_\beta \ t' \quad \Rightarrow \quad \exists t'''.\, t'' \ \rightarrow^{*}_\beta \ t''' \ {}^{*}_{\text{s}}\!\leftarrow \ t'$

Proof. Induction on the number of reduction steps in $t \ \rightarrow^{*}_\beta \ t'$, using the above lemma and then the induction hypothesis. □

Theorem 2.20 (confluence). $(\ \rightarrow_{\text{s}} \ \cup \ \rightarrow_\beta\ )^*$ *is confluent*

Proof. Using Corollary 2.17 we get the following diagram



where the uppermost square is the well-known fact that the untyped λ-calculus is confluent, the left and the right square are lemma 2.19 and the lower square is Corollary 2.11. $\square$

2.3 Normalization by evaluation

Intuitively, normalization by evaluation works by animating all applications and λ-abstractions, but keeping free variables as syntactical objects. This explains the formulation of the following lemma.

Theorem 2.21. *If t is a term without $\downarrow(\cdot)$ with all applications, λ-abstractions and* bound *variables overlined, but all* free *variables beeing syntactical and moreover, after removing all over/underlining t is a normal (ordinary) λ-term, then $\downarrow(t) \;\to_{\mathsf{s}}^{*}\; t'$ where t' is a syntactical representation of t.*

Proof. Induction on the length of t, using the fact that normal terms are given by the grammar $x\vec{t} \mid \lambda x t$.

In the case $x\vec{t}$ show by induction on the length of the $\vec{t}$ that $\underline{x}\vec{t} \;\to_{\mathsf{s}}^{*}\; \underline{x}\vec{t'}$, where $\vec{t'}$ is a syntactical representation of the $\vec{t}$. If $\vec{t}$ is empty, the claim is trivial; otherwise $(\underline{x}\vec{t})\overline{@}t \;\overset{\text{SideIH}}{\to_{\mathsf{s}}^{*}}\; (\underline{x}\vec{t'})\overline{@}t \;\to_{\mathsf{s}}\; (\underline{x}\vec{t'})\underline{@}{\downarrow}(t) \;\overset{\text{IH}}{\to_{\mathsf{s}}^{*}}\; \underline{x}\vec{t}\,\underline{@}t'$.

In the case $\lambda x t$ we have $\downarrow(\overline{\lambda \overline{x}}\, t) \;\to_{\mathsf{s}}\; \underline{\lambda \underline{z}}\, \downarrow(t\,[\underline{x}{:=}\underline{z}]) \;\overset{\text{IH}}{\to_{\mathsf{s}}^{*}}\; \underline{\lambda \underline{z}}\, t'\,[\overline{x}{:=}\underline{z}]$. $\square$

Definition 2.22. For an ordinary λ-term t, let $[\![t]\!]$ be t with all applications, abstractions and bound variables overlined and all free variables underlined.

Immediately from the definiton of $\to_\beta$ we get

Proposition 2.23. $t \to t' \quad \Rightarrow \quad [\![t]\!] \;\to_\beta\; [\![t']\!]$

Corollary 2.24 (untyped normalization by evaluation). *If t is an ordinary (untyped) λ-term that has a normal form, then $\downarrow([\![t]\!])$ reduces to a syntactical representation of the normal form.*

The corollary should be read as follows: we define our semantics of the untyped λ-calculus as equivalence classes the two-level λ-terms with respect to the symmetric-reflexive-transitive closure of the reduction relation. This is a reasonable identification, as only "semantical applications" and the "function $\downarrow(\cdot)$" are animated by the reduction relation. Moreover, by theorem 2.20 we know, that we're not identifying different syntactical objects. In the sense of this semantics we can say that $\downarrow(\llbracket t \rrbracket)$ *is* the normal form of t, if it exists.

3 Implementation

So far, untyped normalization by evaluation was only presented as a theoretical result (using as semantics the two-level λ-terms with the symmetric-reflexive-transitive closure of the reduction relation factored out). For an actual implementation based on interpreting the $\overline{\lambda \cdot} \ldots$ as the function abstraction of the implementation language certain points have to be considered:

- We were considering equivalence classes of α-equal terms and certain reductions envolved the use of a "new" variable.
- As where dealing with untyped λ-terms we loose strong normalization, so reduction strategy becomes relevant to treat terms correctly that are weakly, but not strongly normalizing.
- $\rightarrow_{\mathtt{down}}$ and $\rightarrow_{\mathtt{app}}$ depended on knowing whether an object is syntactical. This should be easy (and efficient!) to test; in particular we do *not* want to inspect the whole term.

3.1 De Bruijn Indices

Although the problem of α-equality and "new" variables could, in principle be solved by using representatives and a "`gensym`-approach", it seems much clearer to use de Bruijn indices [4]. We think of de Bruijn variables as follows: every node has a level, counting the number of λ's passed on the way from the root of the term to that node. A variables "knows" the level at which it is bound and the index representing it is the difference of the level it is bound and the level it appears. To implement this point of view we add to our concrete syntax

$$n \mid [r] \mid rs@$$

(variables, which are natural numbers; λ-abstraction, which is unary when using de Bruijn indices; application, written in postfix notation, for ease of parsing) an "abstract syntax" which are function from levels (i.e. natural numbers) to the concrete representation of a term, when appearing at this level.

Abstraction, application, free variables and variables bound at a certain level can easily be implemented in the abstract syntax:

```
;; Terms
;; -----
;; Terms are mappings from the level to the syntactical
;; representation  of a term at this level:

(define (fv k) (lambda (n) (var (+ n k))))
(define (bv k) (lambda (n) (var (- n k 1))))
(define (term-app termA termB)
  (lambda (n)
    (app (termA n)(termB n))))
(define (term-abstr term)
  (lambda (n)
    (abstr (term (+ n 1)))))
```

Here `var`, `abstr` and `app` are the variables, abstraction and application at the level of the concrete syntax.

3.2 Lazy Evaluation

Faced with the problem of weakly but not strongly normalizing terms we use the well-known fact that left-most-outer-most reduction is a safe reduction strategy, i.e. will find the β-normal form if there is one. (This is an easy corollary of the standardization theorem.) Due to corollary 2.17 this suffices, as $\to_{\mathtt{s}}$ alone *is* strongly normalizing (every reduction step removes overlinings or symbols).

A standard way of achieving this reduction strategy in call-by-value languages like `Scheme` is "delaying" everything, i.e. our "official" objects are functions of no arguments that, when evaluated return the real object.

Remark 3.1 (special case of strongly normalizing terms). It should be noted, that due to the confluence result (Theorem 2.20) *any* reduction strategy would work if we *would* restrict ourselfs to strongly normalizing terms (in other words, for simply typeable λ-terms we're still as flexible as any other variant of normalization by evaluation).

3.3 Syntax versus Functions

To allow easy testing whether an object is a syntactical entity, we first note that everything we ever have to handle either is a function, i.e. somthing starting with $\overline{\lambda}\cdot\ldots$, or a syntactical term. Therefore we use two tags for functions or syntax. Of course, these tags can only be decided *after* calculating the object, i.e. they come after the delay:

```
;; Semantics
;; =========
```

```
;; Each semantical object is a function of no argument,
;; that evaluates to one of the following:
;; - a pair ('fun . function)
;;   where function is from the semantics into the semantics
;; - a pair ('syn . term)
;;   where term is a term of the abstract syntax
```

These two tags can be used to guide the function $\downarrow(\cdot)$ where the substitution is implemented by an actual call of the (semantical) function:

```
;; down takes an object and returns a representation in the
;; abstract syntax

(define (make-bv n)
  ;; the correct representation of a variable bound at level n
  (lambda ()
    (cons 'syn (bv n))))

(define (down obj)
  (let ((val (obj)))         ;; inspect the object
    (if (eq? (car val) 'syn) ;; if it is in the syntax...
        (cdr val)            ;; ...we're done
        (lambda (n)
          ;; otherwise construct something of the abstract syntax
          ((term-abstr
            ;; by a syntactical abstraction
            ;; and application to a ``new'' variable
            (down ((cdr val) (make-bv n)))) n)))))
```

Once we have defined the function `down`, we can define the semantical application that also has to take care of the $\rightarrow_{\mathsf{app}}$ -reduction.

```
;; sem-app takes two objects and applies them to each other

(define (sem-app obja objb)
  (lambda () ;; we delay
    (let ((obj (obja)))          ;; first inspect ``function argument''
      (if (eq? (car obj) 'fun)   ;; if it is a function...
          (((cdr obj) objb))     ;; ...just call
          ;; otherwise we have some syntax, so transform the argument
          ;; into syntax as well and apply them there
          (let ((term (down objb)))
            (cons 'syn (term-app (cdr obj) term)))))))
```

3.4 Syntactical abbreviations

It is easy to write an evaluation function for this semantics; however it is even more convenient to just define appropriate syntactical abbreviations and later "program" as usual in the untyped λ-calculus.

```
(define-syntax lam
  ;; abstraction: remember: functions are delayed ones!
  (syntax-rules ()
    ((_ (arg) . body)
     (lambda () (cons 'fun (lambda (arg)  . body))))
    ((_ (arg . args) . body)
     ;; also support implicit currying
     (lam (arg) (lam args . body)))))

(define-syntax !
  ;; application:
  ;; remember: delayed as well, but don't forget to
  ;;           undelay the result
  (syntax-rules ()
    ((_ fun  arg)
     (lambda () ((sem-app fun arg))))
    ((_ fun arg . args)
     ;; also support implicit uncurrying
        (! (! fun arg) . args))))


(define (? value)
  (newline)
  (display (syntax-to-string ((down value) 0)))
  (newline))
```

`syntax-to-string` is a function that translates a list-representation of the concrete syntax into a string and does *not* do any computation. It should be noted that the `(lam (...)...)`-construct is mapped to the (appropriately delayed and tagged) `lambda`-binder of `Scheme`.

3.5 Examples

We can use Church-style natural numbers and use meta-syntactical abrevations for them.

```
;; Church-style natural numbers
(define ZERO (lam (step base) base))
(define SUC (lam (n step base)
                 (! step (! n step base))))
```

```
(define make-nat (lambda (n)
                   (if (= 0 n) ZERO
                       (! SUC (make-nat (- n 1))))))
```

Using variables...

```
(define (make-var number)
  (lambda () (cons 'syn (fv number))))
```

...the results are as expected:

```
> (? (! (make-nat 2)(make-nat 3)))

[[1 1 1 1 1 1 1 1 1 0 @ @ @ @ @ @ @ @ @]]
> (? (! (make-nat 2)(make-var 42)))

[43 43 0 @ @]
```

But we can also use more general examples, including the non F-typeable `K I` Ω.

```
(define S (lam (x y z) (! x z (! y z))))
(define K (lam (x y) x))
(define I (! S K K))         ;; (? I)             --> [0]
(define w (lam (x) (! x x))) ;; (? w)             --> [0 0 @]
(define Omega (! w w)) ;; You can define it, but better don't
                       ;; look at it via (? ...)
                       ;; However:
                             ;; (? (! K I Omega)) --> [0]
```

4 Conclusion and further work

The idea of normalization by evaluation has been extended to the full untyped λ-calculus. In particular we still get a simple, fully functional algorithm for normalizing untyped λ-terms *without* the need of implementing a substitution. Besides the fact that this is a computational complete system (which system F is not), this work might give some hints on what normalization by evaluation does, and might justify the claim that types are used "for η-expansion only".

An area of further research might be the operational behavior: At the moment efficiency is handled only in the sense that we have termination whenever there is a normal form (but tests show that the runtime of this implementation is comparable with that of other implementations of normalization by evaluation). However the left-most-outer-most reduction strategy might duplicate redexes and therefore cause calculations to be done several times. This problem can be overcome by using operational concepts to cache values previously calculated. It is easy to redefine `sem-app` in such a way, once we allow the use of `set!`.

References

[1] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Reduction-free normalisation for a polymorphic system. In *Proceedings of the Eleventh IEEE Symposium on Logic in Computer Science (LICS '96)*, pages 98–106, 1996.

[2] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In B. Möller and J.V. Tucker, editors, *Prospects for Hardware Foundations*, number 1546 in LNCS, pages 117–137. Springer Verlag, Berlin, Heidelberg, New York, 1998.

[3] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ–calculus. In R. Vemuri, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science (LICS '91)*, pages 203–211, 1991.

[4] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church–Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[5] Felix Joachimski and Ralph Matthes. Standardization and confluence for a lambda calculus with generalized applications. In Leo Bachmair, editor, *Rewriting Techniques and Applications, Proceedings of the 11th International Conference RTA 2000, Norwich, UK*, volume 1833 of *Lecture Notes in Computer Science*, pages 141–155. Springer Verlag, 2000.

[6] René Vestergaard. Normalization by Evaluation for System F. `http://iml.univ-mrs.fr/~vester/Writings/index.html`.